

AFRL-IF-RS-TR-2003-173
Final Technical Report
July 2003



ASPECT ORIENTED PROGRAMMING

Palo Alto Research Center

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J138

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.


The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK


This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-173 has been reviewed and is approved for publication.

APPROVED:


ROBERT J. PARAGI
Project Engineer

FOR THE DIRECTOR:


JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JULY 2003	3. REPORT TYPE AND DATES COVERED Final Aug 97 – Apr 03	
4. TITLE AND SUBTITLE ASPECT ORIENTED PROGRAMMING			5. FUNDING NUMBERS C - F30602-97-C-0246 PE - 62301E PR - F374 TA - 01 WU - 02	
6. AUTHOR(S) Gregor Kiczales, James Hugunin, Erik Hilsdale, Mik Kersten, Jeff Palm, Crista Lopes, Bill Griswold, and Wes Isberg				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Palo Alto Research Center 3333 Coyote Hill Road Palo Alto California 94304-1314			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/ITB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-173	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Robert J. Paragi/ITB/(315) 330-3547/ Robert.Paragi@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) Over the lifetime of the project we developed a general-purpose aspect-oriented programming (AOP) extension to Java, called AspectJ, cultivated a user community for AspectJ, and showed that the technology was useful for a wide range of software development problems. AspectJ is now the de facto standard AOP language, not just for Java, but in some sense for languages beyond Java. This significant milestone came about through major scientific, engineering, and community building accomplishments throughout the life of the project.				
14. SUBJECT TERMS Object-Oriented Programming, Programming Language Aspects, Aspect oriented Programming			15. NUMBER OF PAGES 155	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	
NSN 7540-01-280-5500			Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102	

Table of Contents

1	SUMMARY OF PROJECT RESULTS.....	1
2	AN OVERVIEW OF ASPECTJ.....	3
2.1	Introduction.....	3
2.2	Basic Design Assumptions	4
2.3	The Language.....	5
2.4	Implementation	17
2.5	Understanding Crosscutting Structure	21
2.6	Related Work	24
2.7	Summary	26
2.8	Acknowledgements.....	27
2.9	References.....	27
3	GETTING STARTED WITH ASPECTJ.....	32
3.1	AspectJ semantics	33
3.2	Development Aspects	35
3.3	Production Aspects	38
3.4	Conclusion	44
3.5	Acknowledgements.....	45
3.6	References.....	45
4	THE ASPECTJ™ PROGRAMMING GUIDE	46
4.1	Preface.....	46
4.2	Getting Started with AspectJ	46
4.3	The AspectJ Language.....	61
4.4	Examples.....	75
4.5	Idioms	106
4.6	Pitfalls	107
	APPENDIX A ASPECTJ QUICK REFERENCE	110
A.1	Pointcuts.....	110
A.2	Type Patterns	111
A.3	Advice	112
A.4	Inter-type member declarations	112
A.5	Other declarations	113
A.6	Aspects.....	113
	APPENDIX B LANGUAGE SEMANTICS	115
B.1	Introduction.....	115

B.2	Join Points.....	116
B.3	Pointcuts.....	117
B.4	Advice.....	128
B.5	Static crosscutting.....	133
B.6	Aspects.....	140
APPENDIX C IMPLEMENTATION LIMITATIONS		145
APPENDIX D TALKS PRESENTED		147
D.1	Tutorials.....	147
D.2	Talks, Demos, and Keynotes	147

List of Figures

Figure 1 Join points of AspectJ.....	6
Figure 2 Part of a simple figure editor program.	7
Figure 3 Object and joinpoints created by figure code.	8
Figure 4 Primitive pointcut designators and the rules for what join points they match. ...	9
Figure 5 A portion of the screen when using the AJDE extension to JBuilder 3.5.	20
Figure 6 A portion of the screen when using the AspectJ-aware extension to emacs.	21
Figure 7 UML for Figure Editor Example.....	33
Figure 8 A snapshot of screen when using the AspectJ-aware extension to emacs.	40
Figure 9 Examining the partially re-factored code with the AJDE extension to JBuilder.	43
Figure 10 UML for the FigureEditor example.....	47
Figure 11 Roles for Point.....	80
Figure 12 Telecom interactions	97

1 Summary of Project Results

This is the final report for the Aspect-Oriented Programming project funded under contract #F30602-97-C-0246 (AO# J138). This report along with the material on the attached CD-ROM concludes our obligations under that contract.

Over the lifetime of the project we developed a general-purpose aspect-oriented programming (AOP) extension to Java, called AspectJ, cultivated a user community for AspectJ, and showed that the technology was useful for a wide range of software development problems. AspectJ is now the de facto standard AOP language, not just for Java, but in some sense for languages beyond Java. This significant milestone came about through major scientific, engineering, and community building accomplishments throughout the life of the project.

Specific accomplishments include:

- Development of a clear set of core design elements for general purpose AOP. This includes the orthogonal notions of join points, means of identifying join points (pointcuts) and means of specifying semantics at join points (advice, declare error/warning, and inter-type declarations).
- Development of viable implementation strategies for the above technology, including not just pre-processing approaches, but also byte-code compiler, incremental compiler and load-time weaving approaches.
- Design and implementation of 9 major and 51 minor public releases.
- Development of a vibrant user community, through user mailing lists, workshops, BOFs, user site visits etc. The final users@aspectj.org mailing list had 830 subscriptions, and the final announce@aspectj.org mailing list had 1706.
- Development of documentation and training material including:
 - A 91 page programming guide
 - A 121 slide tutorial
 - A 54 slide “one-hour talk”
- Presentation of the tutorial 18 times since January 2000 to various conference and industry audiences. (See appendix for complete list.) The tutorial was also presented numerous times prior to January 2000.
- Presentation of 25 shorter talks since January 2000. This includes one-hour invited or keynote talks, demos and short tutorials. (See appendix for complete list.) There were also numerous short presentations prior to 2000.
- Publication of 4 papers:
 - Getting started with Aspect J. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William Griswold, Communications of the ACM October 2001, Volume 44 Issue 10
 - Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. An Overview of AspectJ In Proc. Of ECOOP, Springer-Verlag (2001).
 - “A Study on Exception Detection and Handling Using Aspect-Oriented Programming” Martin Lippert and Cristina Lopes. In Proc. of the 22nd

International Conference of Software Engineering (ICSE'2000), Limerick, Ireland. IEEE Computer Society. June 2000.

- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In Proc. of ECOOP, Springer-Verlag (1997).

Note that the 1997 and 2001 ECOOP papers are listed by citeseer.org as being the #4 and #18 citation respectively in their year of publication.

In addition, AspectJ users who are not directly members of the project have:

- Written three books:
- Aspect-Oriented Programming with AspectJ, by Ivan Kiselev
- Mastering AspectJ: Aspect-Oriented Programming in Java, by Joseph D. Gradecki and Nicholas Lesiecki
- AspectJ In Action, by Ramnivas Laddad
- Written numerous papers using or based on AspectJ
- 9 papers at the 2003 Aspect-Oriented Software Development Conference.
- 5 papers at the 2002 Aspect-Oriented Software Development Conference.
- 2 papers at ECOOP 2002
- 2 papers at OOPSLA 2002

And at least three groups have developed plans to offer commercial consulting support for AspectJ.

After discussion with DARPA, the project concluded by transferring the AspectJ code base to IBM's Open Source development community at eclipse.org under the Common Public License. This ensures that AspectJ can remain an Open Source development project. PARC retains patent rights that may apply to AOP beyond AspectJ. PARC also retains the copyright on all training material, but has committed to allow AspectJ project members no longer at PARC rights to use that material in AspectJ books or courses they may prepare or deliver. The members of the AspectJ team also expect to spend a modest amount of time helping Open Source developers become familiar with the AspectJ code base so that a viable external community of developers can get going. This project is now hosted at <http://eclipse.org/aspectj>

A separate CD-ROM has been delivered that contains the major concrete products of this project:

- Documentation and training material
- programmingGuide.pdf - programming guide
- tutorial.ppt - tutorial
- oneHourTalk.ppt - standard one-hour talk
- AspectJ-1.0.6 system
- aspectj-tools-1.0.6.jar - self-installing .jar file
- aspectj-tools-src-1.0.6.jar – tar/gzipped source code
- aspectj-docs-1.0.6.tgz – tar/gzipped documentation and examples
- AspectJ-1.1candidate1 system
- aspectj-1.1rc1.jar - self-installing .jar file
- aspectj-src-1.1rc1.tgz – tar/gzipped source code

2 An Overview of AspectJ

AspectJ is a simple and practical aspect-oriented extension to Java™. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns. Join points are principled points in the execution of the program; pointcuts are collections of join points; advice is a special method-like construct that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprised of pointcuts, advice, and ordinary Java member declarations. AspectJ code is compiled into standard Java bytecode. Simple extensions to existing Java development environments make it possible to browse the crosscutting structure of aspects in the same kind of way as one browses the inheritance structure of classes. Several examples show that AspectJ is powerful, and that programs written using it are easy to understand.

2.1 Introduction

Aspect-oriented programming (AOP) [21] has been proposed as a technique for improving separation of concerns in software.¹ AOP builds on previous technologies, including procedural programming and object-oriented programming, which have already made significant improvements in software modularity.

The central idea in AOP is that while the hierarchical modularity mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Instead, we believe that in the implementation of any complex system, there will be concerns that one would like to modularize, but for which the implementation will instead be spread out. This happens because the natural modularity of these concerns crosscuts the natural modularity of the rest of the implementation.

AOP does for concerns that are naturally crosscutting what OOP does for concerns that are naturally hierarchical—it provides language mechanisms that explicitly capture crosscutting structure. This makes it possible to program crosscutting concerns in a modular way, and thereby achieve the usual benefits of modularity: simpler code, that is easier to develop and maintain, and that has greater potential for reuse. We call such well-modularized crosscutting concerns *aspects*.²

¹ When we say “separation of concerns” we mean the idea that it should be possible to work with the design or implementation of a system in the natural units of concern – concept, goal, team structure etc. – rather than in units imposed on us by the tools we are using. We would like the modularity of a system to reflect the way “we want to think about it” rather than the way the language or other tools force us to think about it. In software, Parnas is generally credited with this idea [37, 38].

² AOP support can be added to languages that are not object-oriented. The key property of an AOP language is that it provides crosscutting modularity mechanisms. So when we add AOP to an OO language, we add constructs that crosscut the hierarchical modularity of OO programs. If we add AOP to a procedural language, we must add constructs that crosscut the block structure of procedural programs [10, 12].

AspectJ is a simple and practical aspect-oriented extension to Java. This paper presents an overview of AspectJ, including the core language features, how the compiler works, development environment support, and several examples of how it can be used. The examples show that using AspectJ we can code, in clear form, crosscutting concerns that would otherwise lead to tangled code.

The main elements of the language design are now fairly stable, but the AspectJ project is not nearly finished. We continue fine-tuning parts of the language, building a third-generation compiler, expanding the IDE support we provide, extending the documentation and training material, and building up the user community. We plan to work with that user community to empirically study the practical value of AOP.

The next section describes the basic assumptions behind the AspectJ language design. Section 2.3 presents the core language. Section 2.4 outlines the compiler. Section 2.4.4 describes the AspectJ-aware tools we have developed. Section 2.5 shows that AspectJ can capture crosscutting structure in elegant and easy to understand ways. We conclude with a discussion of related and future work. By presenting this overview, the paper provides a foundation for detailed discussion. Future papers will present detailed formal semantics, language design rationale, language and tool implementation issues, and analysis of software engineering benefits.

2.2 Basic Design Assumptions

AspectJ is intended to be the basis for an empirical assessment of aspect-oriented programming. We want to know what happens when a real user community uses an AOP language. What kinds of aspects do they write? Can they understand each other's code? What kinds of idioms and patterns emerge? What kinds of style guidelines do they develop? How effectively can they work with crosscutting modularity? And, above all, do they develop code that is more modular, more reusable, and easier to develop and maintain?

Because this is our goal, designing and implementing AspectJ is really just part of the project. We must also develop and support a substantial user community. To make it possible to build a large user community, we have chosen to design AspectJ as a *compatible* extension to Java that will facilitate adoption by current Java programmers. By compatible we mean four things:

- *Upward compatibility* — all legal Java programs must be legal AspectJ programs.
- *Platform compatibility* — all legal AspectJ programs must run on standard Java virtual machines.
- *Tool compatibility* — it must be possible to work with AspectJ using existing tools, including integrated development environments (IDEs), documentation and design tools.
- *Programmer compatibility* — Programming with AspectJ must feel like a natural extension of programming with Java.

The programmer compatibility goal has been responsible for much of the feel of the language. Like Java, AspectJ is a general-purpose rather than domain-specific language;

AspectJ has a Java-like balance between declarative and imperative constructs; AspectJ is statically typed, and uses Java’s static type system. Like Java, we have been relatively conservative about what features to put into AspectJ. AspectJ is a small extension to Java, and programming with AspectJ is a small extension to programming with Java. In AspectJ programs we use the OO constructs to do what they do well, and then use the aspect-oriented constructs to handle the concerns that OO alone cannot effectively modularize.

There are several potentially valuable AOP research goals that AspectJ is not intended to meet. It is not intended to be a “clean-room” incarnation of AOP ideas, a formal AOP calculus or an aggressive effort to explore the AOP language space. Instead, AspectJ is intended to be practical AOP language that provides, in a Java compatible package, a solid and well worked-out set of AOP features.

2.3 The Language

With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns. *Join points* are principled points in the execution of the program; *pointcuts* are a means of referring to collections of join points and certain values at those join points; *advice* is a method-like construct that can be attached to pointcuts; and *aspects* are modular units of crosscutting implementation, comprised of pointcuts, advice, and ordinary Java member declarations.

This section of the paper presents the main elements of the language. In keeping with Java terminology, we call this the ‘static’ part of the language, because it does not involve aspect instances or the open-class mechanism. The presentation is informal and example based.

2.3.1 Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the frame of reference that makes it possible for execution of a program’s aspect and non-aspect code to be coordinated properly.

In previous work, we have used several different kinds of join point models, including primitive application nodes in a dataflow graph [31] and method bodies [26]. Early versions of AspectJ used a model in which the join points were principled places in the source code.

AspectJ is now based on a model in which join points are principled points in the dynamic execution of the program. This model gives us important additional expressive power, discussed in Section 2.3.9.2. AspectJ’s join points can be considered as nodes in a simple runtime object call graph. These nodes include points at which an object receives a method call and points at which a field of an object is referenced. The edges are control flow relations between the nodes. In this model control passes through each join point twice, once on the way in to the sub-computation rooted at the join point, and once on the way back out. The join points of AspectJ are shown in Figure 1.

<i>kind of join point</i>	<i>Points in the program execution at which...</i>
method calls constructor calls*	a method is called (or a constructor of a class is called). Call join points are in the calling object, or from no object if the call is from a static method.
method call receptions constructor call receptions	an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method/constructor has been called.
method executions* constructor executions*	an individual method or constructor is invoked.
field gets	a field of an object, class or interface is read.
field sets	a field of an object or class is set.
exception handler executions*	an exception handler is invoked.

Figure 1 Join points of AspectJ.

Most examples in this paper are based on a simple figure editor, the kernel of which is shown in Figure 2. (Code from the paper is at <http://aspectj.org/home/gregor/ecoop2001.html>.) Based on those classes, executing the first three lines of code in Figure 3 builds the objects shown to the right. In this picture large circles represent objects, square boxes represent methods and small numbered circles represent join points. Executing the last line causes a computation that proceeds through the join points labeled at the right:

1. A method call join point at which the `slide` method is called on the object `ln1`.
2. A method call reception join point at which `ln1` receives the `slide` call.
3. A method execution join point at which the particular `slide` method defined in the class `Line` is called.
4. A field get join point where the `_p1` field of `ln1` is read.
5. A method call join point at which the `slide` method is called on the object `pt1`.
- ...
8. A method call join point at which the `getX` method is called on the object `pt1`.
- ...
11. A field get join point where the `_x` field of point `pt1` is read.
control returns back through join points 11, 10, 9 and 8.

<pre> interface FigureElement { public void slide(int dx, int dy); ... } class Point implements FigureElement { private int _x, _y; public Point(int x, int y) { _x = x; _y = y; } public int getX() { return _x; } public int getY() { return _y; } public void setX(int x) { _x = x; } public void setY(int y) { _y = y; } public void slide(int dx, int dy) { setX(getX() + dx); setY(getY() + dy); } ... } </pre>	<pre> class Line implements FigureElement { private Point _p1, _p2; public Line(Point p1, Point p2) { _p1 = p1; _p2 = p2; } public Point getP1() { return p1; } public Point getP2() { return p2; } public void setP1(Point p1) { _p1 = p1; } public void setP2(Point p2) { _p2 = p2; } public void slide(int dx, int dy) { _p1.slide(dx, dy); _p2.slide(dx, dy); } ... } </pre>
---	---

Figure 2 Part of a simple figure editor program.

12. A method call join point at which the `setX` method is called on `p1`.
... and so on, until control finally returns back through 3, 2 and 1

2.3.2 Pointcut Designators

A pointcut is a set of join points that optionally exposes some of the values in the execution context of those join points. AspectJ includes several primitive *pointcut designators*, which, based on different kinds of criteria, can identify join points of all types. Pointcuts can be composed and new pointcut designators can be defined in terms of those combinations. Pointcuts are not higher order, nor are pointcut designators parametric.

A simple way to think of pointcut designators is in terms of ‘matching’ certain join points at runtime. For example, the pointcut designator `‘receptions(void Point.setX(int))’` matches all method call reception join points at which the receiver is of type `Point` (or a sub-type of `Point`), and the Java signature of the method call is `‘void setX(int)’`. Intuitively, this refers to every time a point receives a call to change its x coordinate. Similarly `‘receptions(void FigureElement.slide(int, int))’` intuitively refers to every time any kind of figure element (i.e. an instance of `Point` or `Line`) receives a call to slide a certain distance.

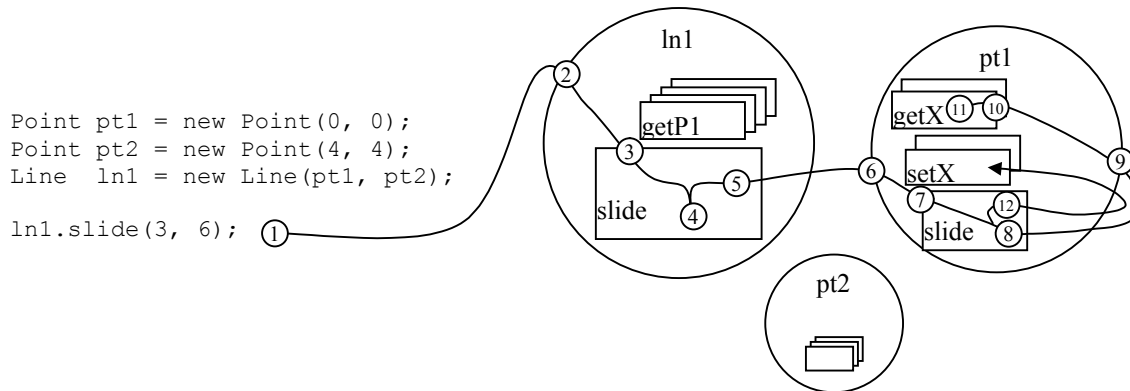


Figure 3 Object and joinpoints created by figure code.

Pointcuts can be combined using special and, or and not operators ('&&', '||' and '!'). The following compound pointcut designator refers to whenever a Point receives a call to change its x or y coordinate.

```
receptions(void Point.setX(int)) ||
receptions(void Point.setY(int))
```

2.3.2.1 Primitive Pointcut Designators

AspectJ includes a variety of primitive pointcut designators. These work in different ways to identify join points. Some primitive pointcut designators only identify pointcuts of one kind, for example `receptions` only matches method call reception join points. Others match any of the nine kinds of join points at which a certain property holds. For example, '`instanceof(Point)`' matches all join points at which the currently executing object (the value of '`this`') is an instance of `Point` or a subclass of `Point`.³

These two kinds of join point designators can be combined to identify join points in crosscutting ways. For example:

```
!instanceof(FigureElement) &&
calls(void FigureElement.slide(int, int))
```

matches all method calls to `slide` that do not come from an object that is a figure element.⁴

The primitive pointcut designators are summarized in Figure 4. They are explained further as they are used in the paper.

³ The name `instanceof` is chosen because of the similarity in semantics to Java's `instanceof` operator.

⁴ This will match calls that come from static methods. Since there is no currently executing object in such methods, `instanceof(FigureElement)` will not match such join points.

<pre>calls(<i>signature</i>) receptions(<i>signature</i>) executions(<i>signature</i>)</pre> <p>Matches call/reception/execution join points at which the method or constructor called matches <i>signature</i>.</p> <p>The syntax of a method signature is:</p> <pre><i>ResultTypeName ReceiverTypeName.method_id(ParameterTypeName, ...)</i></pre> <p>The syntax of a constructor signature is:</p> <pre><i>NewObjectTypeName.new(ParameterTypeName, ...)</i></pre>
<pre>gets(<i>signature</i>) gets(<i>signature</i>) [<i>val</i>] sets(<i>signature</i>) sets(<i>signature</i>) [<i>oldVal</i>] sets(<i>signature</i>) [<i>oldVal</i>] [<i>newVal</i>]</pre> <p>Matches field get/set join points at which the field accessed matches the signature.</p> <p>The syntax of a field signature is:</p> <pre><i>FieldName ObjectTypeName.field_id</i></pre>
<pre>handles(<i>ThrowableTypeName</i>)</pre> <p>Matches exception handler execution join points at which the exception handled is of the specified type.</p>
<pre>instanceof(<i>CurrentlyExecutingObjectTypeName</i>) within(<i>ClassName</i>) withincode(<i>signature</i>)</pre> <p>Matches join points of any kind at which the currently executing:</p> <ul style="list-style-type: none"> - object is of type <i>CurrentlyExecutingObjectTypeName</i> - code is contained within <i>ClassName</i> - code is contained within the member defined by the method or constructor signature
<pre>cflow(<i>pointcut_designator</i>)</pre> <p>Matches join points of any kind that occur within the dynamic extent of any join point matched by <i>pointcut_designator</i>.</p>
<pre>callto(<i>pointcut_designator</i>)</pre> <p>Matches method call join points that in one step lead to any reception or execution join points matched by <i>pointcut_designator</i>.</p>

Figure 4 Primitive pointcut designators and the rules for what join points they match.

Any *...TypeName* position does normal sub-type matching. Any *...id* position does matching by string equality. See section 2.3.9.1 for information about more sophisticated wild card matching in these positions.

2.3.2.2 User-defined Pointcut Designators

User-defined pointcut designators are defined with the `pointcut` declaration. The declaration:

```
pointcut moves() :
    receptions(void FigureElement.slide(int, int)) ||
    receptions(void Line.setP1(Point)) ||
    receptions(void Line.setP2(Point)) ||
    receptions(void Point.setX(int)) ||
    receptions(void Point.setY(int));
```

defines a new pointcut designator, `moves()`, that identifies whenever a figure element receives a call of a method that can move it. User-defined pointcut designators can be used wherever a pointcut designator can appear.

2.3.3 Advice

Advice is a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut. AspectJ supports *before*, *after*, and *around* advice. Additionally, there are two special cases of after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point. This advice framework is modeled after the declarative method combination mechanism in CLOS [6, 7, 20, 42] (which itself was modeled on the demon methods of Flavors [8]).

Advice declarations define advice by associating a code body with a pointcut, and a time, relative to each join point in the pointcut, when the code should be executed. The advice declaration

```
static after(): moves() {
    flag = true;
}
```

defines static after advice on the pointcut `moves()`. The ‘`()`’ between ‘`after`’ and the ‘`:`’ means the advice has no parameters. The effect of this declaration is to ensure that the `flag` variable is set to `true` whenever a figure element finishes handling a move method call. The declaration of the variable is shown in the example in Section 2.3.4.)

A simple model for the behavior of advice is in terms of runtime dispatch. (Section 2.4 outlines the techniques the compiler uses to ensure that most if not all of the matching overhead happens at compile time.) Upon arrival at a join point, all advice in the system are examined to see whether any apply at the join point. Any that does is collected, ordered according to specificity (described in Section 2.3.4), and executed as follows:

1. First, any around advice are run, most-specific first. Within the body of an around advice calling `runNext()` invokes the next most specific piece of around advice, or, if no around advice remain, goes to the next step.⁵
2. Then all `before` advice are run, most-specific first.
3. Then the computation proceeds forward from the join point.
4. Execution of `after returning` and `after throwing` advice depends on how that computation terminates.
 - If the computation terminates normally, all `after returning` advice are run, least specific first.
 - If the computation terminates by throwing an exception, all `after throwing` advice are run, least specific first.
5. Then all `after` advice are run, least-specific first.
6. Once all `after` advice run, the return value from step 3, if any, is returned to the innermost call to `runNext` from step 1, and that piece of around advice continues running.
7. When the innermost piece of around advice returns, it returns to the surrounding around advice.
8. When the outermost piece of around advice returns, control continues back from the join point.

Advice can be static or non-static. Static advice runs at any join point matched by its pointcut designator. (Non-static advice only runs in the context of an associated aspect instance.)

2.3.4 Aspects

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a form similar to that of class declarations. Aspect declarations may include pointcut declarations, advice declarations, as well as all other kinds of declarations permitted in class declarations.

The following declaration defines an aspect that implements the behavior of keeping track of whether a figure element has moved recently. This aspect might be used by the screen update mechanism to find out whether anything has changed since the last time the screen was updated. (More sophisticated versions of this aspect will be presented in later sections.)

⁵ A restriction imposed by the Java virtual machine is that no around or before advice can be defined on constructor call reception or constructor execution join points. For most cases where the programmer wants this functionality advice defined on constructor call join points has the same effect.

```

aspect MoveTracking {

    static boolean flag = false;

    static boolean testAndClear() {
        boolean result = flag;
        flag = false;
        return result;
    }

    pointcut moves():
        receptions(void FigureElement.slide(int, int)) ||
        receptions(void Line.setP1(Point)) ||
        receptions(void Line.setP2(Point)) ||
        receptions(void Point.setX(int)) ||
        receptions(void Point.setY(int));

    static after(): moves() {
        flag = true;
    }
}

```

Static advice of an aspect are similar to static methods in that they have access to static members of the class. So in the case the static after advice can reference the static variable `flag`.

2.3.5 Aspect Precedence

In general, more than one piece of advice may apply at a join point. The different advice can come from different aspects or even the same aspect.

In order for large AspectJ programs to work in predictable ways, the relative precedence of such advice must be well defined. The basis for advice precedence in AspectJ is based on the fact that aspects are the primary unit of crosscutting functionality. So advice precedence is resolved with respect to the relative precedence of the aspects in which they are defined.

For two pieces of advice, a_1 and a_2 , defined in aspects A_1 and A_2 respectively, the relative specificity is determined as follows:

- If A_1 and A_2 are the same, whichever piece of advice appears first in that aspect declaration's body is more specific.
This rule exists because one aspect may need to define multiple advice that apply at the same join point. This commonly happens when there are matching before and after advice, but it can also happen with two pieces of advice of the same kind.
- If A_1 directly or indirectly extends A_2 , then a_1 is more specific than a_2 .
This rule is a natural extension of method overriding rules in OO languages. It supports the common case where the related advice are defined in aspects that naturally exist in an extends relationship. (Section 2.3.8 discusses aspect inheritance and overriding in more detail.)

- If A_1 includes a 'dominates' modifier that mentions A_2 , then a_1 is more specific than a_2 .
In some cases, the programmer needs to control precedence between aspects that do not exist in an extends relationship.
- In all other cases the relative specificity between a_1 and a_2 is undefined.
A common case is that two conceptually and semantically independent aspects may define advice that apply at the same join point. In such cases the programmer does not need to control the relative ordering of advice they contain.

The following mobility aspect is an example of the use of the 'dominates' modifier. This simple aspect implements a global flag that freezes all figure elements so that they cannot move. The aspect works by checking the flag before any move operation, and simply doing a "quiet abort" of the operation if moves are disabled.

```
aspect Mobility dominates MoveTracking {
    private static boolean enableMoves = true;

    static void enableMoves() { enableMoves = true; }
    static void disableMoves() { enableMoves = false; }

    static around() returns void: MoveTracking.moves() {
        if ( enableMoves )
            runNext();
    }
}
```

It would not make sense for this aspect to extend `MoveTracking`, because it doesn't define a more specialized version of the move tracking functionality. But it is essential that it have precedence over `MoveTracking`, so that it can abort a move before it gets registered. Note that the code for this aspect shows that one aspect can refer to a pointcut defined in another aspect in the same way that static fields are referred to in Java.

2.3.6 Pointcut Parameters

In many cases it is useful for advice to have access to certain values that are in the execution context of the join points. For example, a more sophisticated version of the move tracking aspect might record the specific figure elements that have moved recently rather than just a single bit saying that some figure element has moved recently.

AspectJ provides a parameter mechanism that makes it possible for advice to see a principled subset of the values in play at join points. This mechanism operates in both advice and pointcut declarations. In advice declarations values can be passed from the pointcut designator to the advice. In pointcut declarations values can be passed from the constituent pointcut designators to the user-defined pointcut designator. In both cases, the flow of values is from the right of the ':' to the left. The net effect is that values made available by primitive pointcut designators can be used in the body of advice.

For example, the following piece of advice has access to both the object receiving the method call and the argument to that call.

```

static before(Point p, int nval): receptions(void p.setX(nval)) {
    System.out.println(
        "x value of " + p + " will be set to " + nval + "." );
}

```

The parameter mechanism uses a combination of positional and by-name matching. The list of parameters to the left of the ‘:’ declares that this piece of advice has two parameters, of type `Point` and `int`, named `p` and `nval` respectively. Then, to the right of the colon, those two parameter ids can be used in the same position that a type name would normally appear, to say that the parameter should get the corresponding value. So, the ‘`p`’ and ‘`nval`’ in ‘`p.setX(nval)`’ mean that the effective signature is `Point.setX(int)`; and that `p` should get the object receiving the call and `nval` should get the value of the first argument to the call.

Definition and use of parameters works in a similar way in user-defined pointcuts. In this code

```

pointcut slides(FigureElement fe):
    receptions(void fe.slide(int, int));

static after(FigureElement figElt): slides(figElt) {
    <in advice body 'figElt' is bound to the figure element>
}

```

the pointcut declaration says that `slides(FigureElement)` exposes a single parameter, of type `FigureElement`, and that it is the receiver of the `slide` method call. The advice declaration says that `figElt` should be bound to the first parameter of `slides`, which is the figure element being slid. Note that the name of the parameter in the pointcut declaration does not have to be the same as within the advice declaration.

Values can be exposed from other primitive pointcut designators as well. A common case is to use `instanceof` with a parameter, to provide access to the object making a call.

```

pointcut gets(Object caller):
    instanceof(caller) &&
    (calls(int Point.getX()) ||
     calls(int Point.getY()) ||
     calls(Point Line.getP1()) ||
     calls(Point Line.getP2()));

```

The primitive pointcut designators expose values as suggested by the naming convention in Figure 4. The `ReceiverTypeName` position in method signatures exposes the object receiving the method call and so on.

2.3.6.1 Static Typing of Receiver

In a *highly polymorphic* pointcut designator like `moves`, there is no common super type that accepts all of the method calls in the pointcut (i.e. there is no type that accepts all of `slide`, `setP1`, `setP2`, `setX` and `setY`). That means it isn’t possible to write moves to expose the figure element that is moving by simply plugging a common parameter into the receiver position of each `receptions`. One cannot write something like:

```
pointcut moves(FigureElement fe):
    receptions(void fe.slide(int, int)) ||
    receptions(void fe.setP1(Point)) ||
    receptions(void fe.setP2(Point)) ||
    ...
```

because `setP1` is not defined on `FigureElement`. Instead, the object receiving the calls must be picked up using `instanceof` as follows:

```
pointcut moves(FigureElement fe):
    instanceof(fe) &&
    (receptions(void FigureElement.slide(int, int)) ||
    receptions(void Line.setP1(Point)) ||
    receptions(void Line.setP2(Point)) ||
    receptions(void Point.setX(int)) ||
    receptions(void Point.setY(int)));
```

2.3.6.2 Access to Return Values

In some cases, after returning advice may want to access the value being returned through the join point. This is done with special syntax, to make it clear that the return value is only present in after returning advice

```
static after(Point p) returning (int x): receptions(int
p.getX()) {
    System.out.println(
        p + " returned " + x + " from getX().");
}
```

2.3.6.3 Parameters and runNext

Within an around advice that has parameters, `runNext` accepts parameters with the same signature as the around advice itself. Calling `runNext` with different actual values for those parameters will cause all remaining advice and the rest of the computation to see the new values. This can be used to implement advice that does pre-processing on the values as follows:

```
static around(int nv) returns void:
    receptions(void Point.setX(nv)) ||
    receptions(void Point.setY(nv)) {
    runNext(Math.max(0, nv));
}
```

The effect of this advice is to ensure that any method call to change the x or y coordinate of a point has its parameter clipped to zero before the change proceeds.

2.3.7 Reflective Access to Join Point

To make certain kinds of advice easier to write, AspectJ provides simple reflective access to information about the current join point. Within the body of an advice declaration, the special variable `thisJoinPoint` is bound to an object representing the current join point. The join point object provides information common to all join points, such as what kind of join point it is and the signature of the surrounding method. It also provides

information specific to each kind of join point, i.e. a field reference join point provides access to the field signature.

2.3.8 Inheritance and Overriding of Advice and Pointcuts

To support aspect-libraries, AspectJ provides a simple mechanism of pointcut overriding and advice inheritance. To use this mechanism a programmer defines an *abstract aspect*, with one or more *abstract pointcuts*, and with advice on the pointcut(s). This, then, is a kind of library aspect that can be parameterized by aspects that extend it. For example, the following defines a simple library of tracing functionality.

```
abstract aspect SimpleTracing {6

    abstract pointcut tracePoints();

    static before(): tracePoints() {
        printMessage("Entering", thisJoinPoint);
    }
    static after(): tracePoints() {
        printMessage("Exiting", thisJoinPoint);
    }

    static void printMessage(String when, JoinPoint tjp) {
        code to print an informative message
        using information from the join point
    }
}
```

To use the library aspect in a specific situation just requires extending the aspect and supplying a concrete definition for the abstract pointcut.

```
aspect SlideTracing extends SimpleTracing {

    pointcut tracePoints():
        receptions(void FigureEditor.slide(int, int));

}
```

Concretizing the abstract pointcut in the sub-aspect has the effect of inheriting the aspect declaration from the super-aspect into the sub-aspect. If the sub-aspect includes a `dominates` modifier, that modifier affects the precedence of the inherited advice.

2.3.9 Property-Based Crosscutting

The pointcuts presented above are all defined in terms of an explicit enumeration of method signatures. Although this is appropriate in many cases, we have found that it is useful to be able to define a pointcut by means of certain other properties of join points. To enable such property-based crosscutting AspectJ includes two kinds of features, wildcarding in pointcut designators and control-flow based pointcut designators (it would

⁶ The syntax of this feature of the language is under revision. The functionality has been present for over a year and will continue to be present in future releases.

be nice to name the more general category that cflow, callto, dflow and all such things are instances of).

2.3.9.1 Wildcarding in Pointcut Designators

AspectJ includes a very simple wildcarding mechanism in pointcut designators. Examples of what this mechanism allows the programmer to say are:

```
receptions(* Point.*(..))
```

Matches receptions of calls to any method defined on the class `Point` (i.e. `slide(int, int)`, `getX()`, `getY()`, `setX(int)`, `setY(int)`).⁷

```
receptions(Point.new(..))
```

reception of a call to any constructor for an object of type `Point` (i.e. the `Point(int, int)` constructor).

```
receptions(public * com.xerox.scanner.*.*(..))
```

reception of call to any public method of an object of any type in the `com.xerox.scanner` package.

```
receptions(* Point.get*())
```

reception of call to any method defined on `Point` for which the the id starts with 'get' and which accepts zero arguments – i.e. the nullary getters `getX()` and `getY()`

2.3.9.2 Control-Flow Based Crosscutting

AspectJ also includes two primitive pointcut designators that permit picking out join points based on whether they are in a particular control-flow relationship with other join points. In order to do this, these designators differ from others in that they accept pointcut designators as parameters.

The `cflow(pcd)` pointcut designator matches all join points that are within the control flow of the join points matched by *pcd*. The points matched by *pcd* itself are not matched `cflow(pcd)`. A canonical use of `cflow` is to distinguish between top-level versus recursive calls of a method. So, for example,

```
pointcut moves(FigureElement fe): <as above>;
```

```
pointcut topLevelMoves(FigureElement fe):  
    moves(fe) && !cflow(moves(FigureElement));
```

The definition of `topLevelMoves` reads as any join point matched by `moves`, but not within the control flow of `moves`. In other words, if the move operation invokes another move operation recursively, that recursive operation will not be matched.

2.4 Implementation

⁷ This will also match calls to methods defined in the class `Object`. If the programmer explicitly wants to exclude these they could write: `receptions(* Point.*(..)) && !receptions(* Object.*(..))`.

This section briefly outlines the current language implementation. All language features described here have been implemented and released, many for six months or more.

The main job of any AspectJ implementation is to ensure that aspect and non-aspect code run together in a properly coordinated fashion. This coordination process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. As is the case with most other language features, aspect weaving can be done by a special pre-processor [19, 26, 31, 35], during compilation, by a post-compile processor [35, 36], at load time, as part of the virtual machine, using residual runtime instructions, or using some combination of these approaches.

The AspectJ language design strives to be silent on the issue of when aspect weaving should be done. We provide a compiler-based implementation of the language that does almost all weaving work at compile-time. This exposes as many programming errors as possible at compile time and avoids unnecessary runtime overhead. Certain special cases of advice involve residual dispatch overhead at runtime.

The compiler uses a pay-as-you-go implementation strategy. Any parts of the program that are unaffected by advice are compiled just as they would be by a standard Java compiler.

The compiler transforms the source program in three ways: the body of every advice declaration is compiled into a standard method, parts of the program where advice applies are transformed to insert static points corresponding to the dynamic join points, and code to implement any residual dynamic dispatch is inserted at those static points.

2.4.1 Compilation of Advice Bodies

Every advice body is compiled into a standard method and the advice is run by a call to the method from appropriate points in the code. This potentially means that the use of advice will add the overhead of a single method-call. But, for static advice, these methods are themselves static so they can be easily in-lined by most JVMs. This means there should generally be no observable performance overhead from these additional method calls. (For non-static advice, the methods are not static, but may be final or called only from monomorphic call sites, and so will still be in-lined by the best virtual machines [17].)

2.4.2 Corresponding Method

The compiler transforms the source program into a form in which there is an explicit *corresponding method* for each dynamic join point that might have advice at runtime. This transformation is only performed for join points that might have advice, not all join points. So, for example, in a program that has advice on the pointcut designated by `gets(int Point.x)`, the compiler would transform references of the form `p.x`, where `p` is a `Point` to `Point.jp0$(p)`, and add the following method to the class `Point`:


```
private static int Point$jp$0(Point obj) {  
    return obj._x;  
}
```

Once this corresponding method has been generated, before and after advice are implemented by making the corresponding method call the advice methods as needed.

There are many cases, including this one, where the compiler will add additional method calls in order to create corresponding methods. This happens for method call, method call reception and field access join points. Extra method calls are also added as part of the implementation strategy for around advice. The overhead of these methods is small in any JVM, and again since they are nearly all static, final or monomorphic, they will be optimized away by good JVMs. We expect a future version of the AspectJ compiler to provide optimizing modes that will eliminate some of these minor overheads.

2.4.3 Dynamic Dispatch

The use of certain pointcut designators, like `cflow`, `callsto`, and `instanceof`, can require a run-time test to determine whether a particular corresponding method actually matches a particular join point designator. In such cases, the corresponding method includes residual testing code that guards the execution of the advice. This overhead is relatively small close to the efficiency of a hand-coded solution to the same design problem. (Dynamic tests of this kind are also involved in dispatching of non-static advice.)

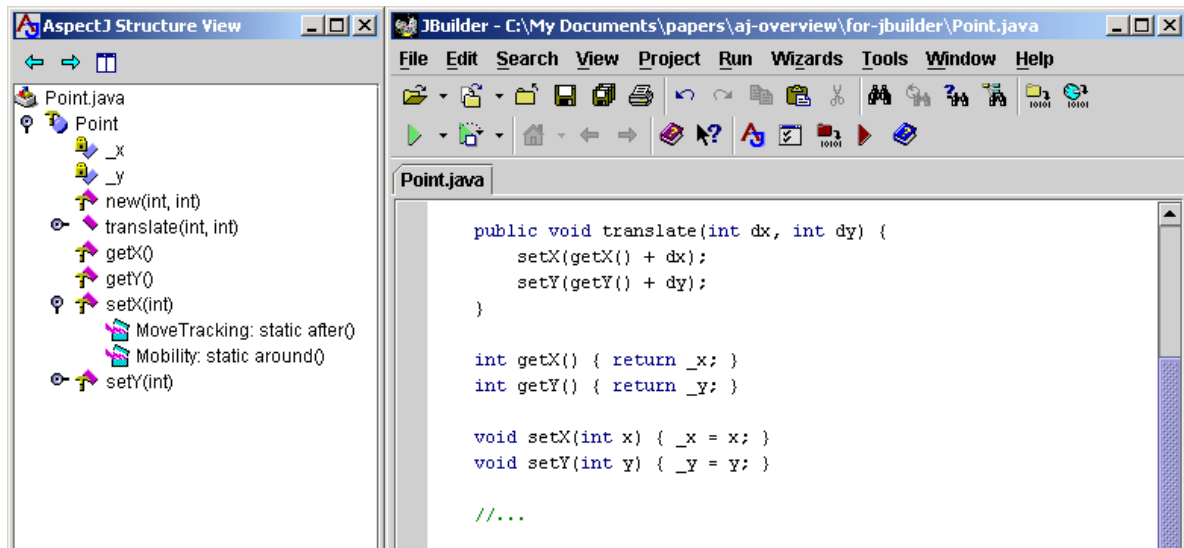


Figure 5 A portion of the screen when using the AJDE extension to JBuilder 3.5.

The main window on the right shows the code for the class `Point`. The structure view on the left shows the class `Point`, and shows that `slide`, `setX` and `setY` are all crosscut by advice; `setX` is further expanded to show what advice crosscuts them. The user can click on the advice to jump there.

2.4.4 Tool Support

In object-oriented programming, development tools typically allow the programmer to easily browse the class structure of their programs. Such support enables the programmer to see a the inheritance and overriding structure in their program, as well as seeing compact representations of the contents of individual classes [14, 16, 39].

For AspectJ, we are developing analogous support for browsing aspect structure. This enables the programmer to see the crosscutting structure in their programs. It works by showing a bi-directional coupling between aspects (and their advice), and the classes (and their methods) that they affect. Figure 5 shows one of the extensions we have made to JBuilder 3.5. This extension to the structure view tool allows the programmer to easily see a summary of the crosscutting into the class `Point`. If the structure view window is focused on an aspect, it will show all the targets of that aspect's advice.

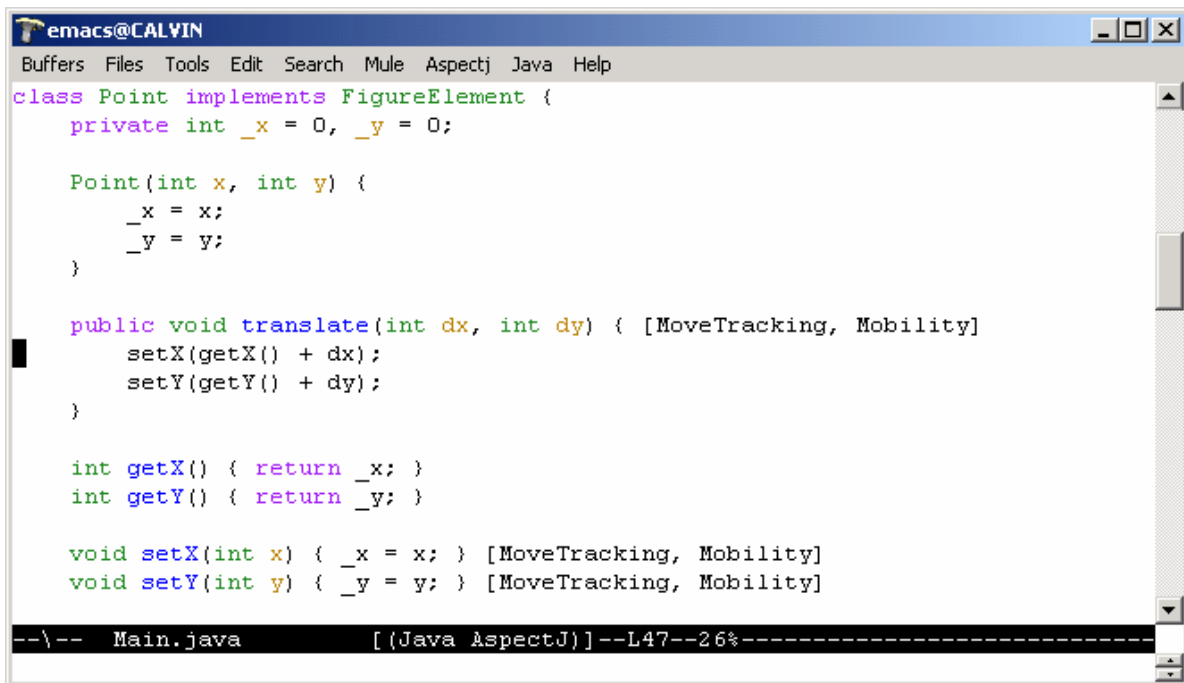


Figure 6 A portion of the screen when using the AspectJ-aware extension to emacs.

The text in [Square Brackets] following the method declarations is automatically generated, and serves to remind the programmer of the aspects that crosscut the method.

A second kind of environment extension provides a more light-weight reminder of the aspect structure. This extension works by annotating the source code, as seen in the editor, with an indication of whether aspects crosscut that code. Figure 6 shows how we have extended emacs with this functionality. The automatically generated annotations name the aspects that crosscut the method. A keystroke command can be used to pop up a menu of the advice, choose one, and jump to it.

We currently support AspectJ-aware extensions to emacs, JBuilder and Forte for Java. Additional tool support includes debugger extensions to understand that advice, display it correctly on the stack etc., as well as extensions to Javadoc [13] to make it understand crosscutting structure and generate appropriate hyper-links etc.

All of these extensions work by consulting a database that is maintained by the compiler. Once the API stabilizes, we intend to make it public so that others can develop tools that use it as well.

2.5 Understanding Crosscutting Structure

One of the most important questions we must answer is how easy is it to program with AspectJ. In particular, is crosscutting structure, implemented with AspectJ, something that appears easy to understand and work with? We do not yet have enough experience to say for sure, but our experience to date suggests that the answer is yes.

2.5.1 Modular, Concise and Explicit

Consider the following simple aspect, which is not part of the figure editor example. This aspect implements a simple error logging functionality, in which every public method defined on any type in the `com.xerox.printers` package logs any errors it throws back to its caller.

```
aspect SimpleErrorLogging {
    static Log log = new Log();

    pointcut publicInterface(Object o):
        instanceof(o) &&
        receptions(public * com.xerox.printers.*.*(..));

    static after(Object o) throwing (Error e): publicInterface(o) {
        log.write(e);
    }
}
```

This aspect appears to be better than the plain Java version of the functionality in several ways:

- The aspect is *more modular* than the plain Java version. In the ordinary Java implementation, every public method would have to inline the logging code itself.
- The aspect is *more concise* than the plain Java version. In the plain Java version something like six lines of code would be added to each public method to wrap the body in a “try... catch...” statement.
- The aspect is *more explicit* than the plain Java version. In this code, the *structural invariant* underlying the crosscutting is clear. A quick look at the code is all it takes to understand that all the public methods defined in the `com.xerox.printers` package should do error logging.

Consider the experience of reading this program if someone else had written it. In the plain Java version, you would see the logging code one method at a time. After seeing a few methods with logging you might guess the logging was being done by all public methods of that class or perhaps even the package. But you would have to use a tool like `grep` to be sure.

In the AspectJ version when you look at the first public method you would see an annotation, something like that in Figure 6, which would tell you that the method was crosscut by the `SimpleErrorLogging` aspect. You could quickly go to the aspect, read the ten lines of code, and understand “what is going on” – the structural invariant would be clear.

Aspects that define crosscutting in terms of explicit enumeration are also more modular, concise and explicit. Consider the now familiar moves `pointcut`:

```

pointcut moves(FigureElement fe):
    instanceof(fe) &&
    (receptions(void FigureElement.slide(int, int)) ||
     receptions(void Line.setP1(Point)) ||
     receptions(void Line.setP2(Point)) ||
     receptions(void Point.setX(int)) ||
     receptions(void Point.setY(int)));

```

Even though this pointcut is enumeration based, putting the complete set of method signatures in a single place makes the crosscutting structure explicit in a clear way. When reading the MoveTracking aspect it is easy to tell what invariant it preserves – whenever something moves it records that fact. Writing the Mobility aspect in terms of MoveTracking.moves, makes it clear that multiple aspects of the implementation crosscut all the move operations. The IDE support ensures that when we happen to be looking at the setX method for Point, we see that Mobility and MoveTracking crosscut there. Navigating to either aspect will show their structure and the fact that Mobility is defined in terms of MoveTracking.

This clarity is preserved when enumeration-based crosscutting is used together with property-based crosscutting. This is evident in the TopLevelMoves pointcut.

```

pointcut topLevelMoves(FigureElement fe):
    moves(fe) && !cflow(moves(FigureElement));

```

Our experience is that the cflow pointcut designator takes only a short while for people learning AspectJ to learn, and once they do so, they find it quite easy to understand this code. It is certainly much easier than to understand what is going on, from the middle of the classic tangled implementation of this functionality.

Clear explicit crosscutting structure can come from the way multiple advice declarations interact as well. In the SimpleTracing aspect of Section 2.3.8, there are two advice declarations:

```

static before(): tracePoints() {
    printMessage("Entering", thisJoinPoint);
}
static after(): tracePoints() {
    printMessage("Exiting", thisJoinPoint);
}

```

Even without knowing what join points tracePoints will match, we understand something important about the structure of this code – the entering and exiting messages happen in pairs, on the way into and back out of join points matched by tracePoints.

2.5.2 The Role of IDE Technology

IDE technology plays an important role in these scenarios. In the course of preparing the paper we encountered a bug in which MoveTracking and Mobility had inconsistent moves pointcuts. The bug was immediately apparent, because the environment showed numerous methods with [Mobility, MoveTracking] and one method with just [MoveTracking].

Because it is now standard practice for OO programmers to use some kind of IDE support—at least in navigating an unfamiliar system—and because it is so easy to

incorporate our technology into an IDE, a programmer can be expected to have IDE support available for such scenarios.

The ability of the IDE to present the structure of the program depends on the degree to which the code declaratively captures that structure. OO IDEs do a good job of presenting inheritance structure because code in OO programming languages captures inheritance explicitly. The AspectJ IDE support works well because code in AspectJ captures crosscutting explicitly.

2.6 Related Work

In earlier work we proposed aspect-oriented programming [21] and presented three examples of domain-specific [1-3] AOP languages [19, 25, 26, 31] that we had developed. AspectJ differs from those three systems in that it is a general-purpose language, it is integrated with Java, it has a dynamic join point model, and we are developing a full compiler, rather than just a pre-processor.

2.6.1 Other Work in AOP

Adaptive Programming [24] provides a special-purpose declarative language for writing class structure traversal specifications. Using this language prevents knowledge of the complete class structure from becoming tangled throughout the code. Adaptive Components [32] build on adaptive programming by using similar graph-language techniques to allow flexible linking of aspectual components and classes. This makes aspectual components reusable. AspectJ supports reusable aspects using the pointcut-overriding and advice-inheritance mechanism, neither of which require a special graph language.

Composition Filters [4, 5] wrap objects inside of filters that operate on the messages the objects receive. The filters have crosscutting access to the messages received by an object. But attachment of filters to objects is done as part of class definitions, so composition filters are less well suited than AspectJ for crosscuts that involve more than one class.

De Volder has proposed a logic meta-programming (LMP) approach that can serve as kind of an AOP language toolkit [11]. In this approach, the equivalent of our pointcut designators use logical queries to specify crosscuts. This approach can take advantage of unification to define parametric pointcut designators. It supports higher-order pointcut designators as well. We have considered extending AspectJ with this kind of power, but have decided not to do so, in order to keep the language simpler and easier for Java developers to learn quickly. We may re-consider this issue in release 2.0 or later; we believe the current pointcut designator syntax leaves us room to do so in an upward compatible way.

2.6.2 Multi-Dimensional Separation of Concerns

Subject-oriented programming is a means for composing and integrating disparate class hierarchies (subjects), each of which might represent different concerns [35]. More recent work on multi-dimensional separation of concerns (MDSOC) [44] is intended to separate concerns along multiple dimensions at once. Hyper/J [36] is a specific proposal for MDSOC. Hyper/J works by having the programmer write two kinds of meta-declarations: the first describes how to slice concerns out of a set of classes; the second describes how to re-compose those concerns into a new program. Hyper/J has the potential to slice a concern out of code without re-factoring the classes. By comparison, in AspectJ the separation of crosscutting concerns is done in the original code, by writing it as an aspect. We believe re-factoring the code with an aspect will be easier to maintain than slicing concerns out, but it is too soon to know.

2.6.3 Reflection

Computational reflection [40, 41] enables crosscutting programs. For example, it is possible to write a small piece of meta-code that runs for all methods. Smalltalk-76 included meta-level functionality [15]. CommonLoops and 3-KRS proposed different meta-level architectures for OO languages [7, 27] PCL provided the first efficient metaobject-protocol [22]. Much of the research in reflection has explored varying the meta-level architecture to support different kinds of crosscutting [27-30, 34, 46] and to achieve flexibility without sacrificing performance [18].

With the exception of reflective access to `thisJoinPoint`, AspectJ has been designed so that the semantics of advice is not a meta-programming nor a reflective semantics. In particular, AspectJ the identifiers in pointcut designators do not refer to program representation or interpreter state – they do involve reification.

2.6.4 Object-Oriented Programming

Flavors [8], New Flavors [33], CommonLoops [7] and CLOS [42] all support multiple-inheritance, declarative method combination and open classes. C++ supports multiple inheritance [43]. AspectJ provides more powerful and modular support for crosscutting than can be achieved with these features.

Ordinary declarative method combination is not sufficient for AOP, because it lacks the wildcarding and control-flow based features that enable property-based crosscutting.

Ordinary multiple-inheritance (MI) is not sufficient for AOP for two reasons. First, a single aspect can include advice for all the different participants in a multi-class interaction. Using MI, a separate mixin-class must be defined for each participant class. Second, aspects work by ‘reverse-inheritance’ – the aspect declares what classes it should affect rather than vice-versa. This means that adding or removing aspects from the system does not require editing affected class definitions.

Completely unstructured open classes, as in CLOS and its ancestors, enable crosscutting modularity, but they do so in a totally unstructured way. In AspectJ, classes and aspects

are modular units, even if an aspect can crosscut classes.⁸ (AspectJ includes an open class mechanism similar to that recently proposed in [9] .)

2.6.5 Other Work

Walker and Murphy have proposed a system based on implicit context that is also intended to improve separation of concerns [45]. Implicit context is similar to AspectJ in that the separation is made explicit in the source code. But, it differs from AspectJ in that it provides reflective access to the entire call history of a system. Thus explicit context can reason about a wider dynamic context than is possible with `cflow`. AspectJ programmers could write aspects to manually gather call history information and thereby duplicate some explicit context functionality.

Implicit Parameters provides dynamically scoped variables within a statically typed Hinley Milner framework [23]. Implicit parameters are lexically distinct from regular identifiers, and are bound by a special `with` construct whose scope is dynamic, rather than static as with `let`. Implicit parameters have some of the power of using `cflow` to pass dynamic context. Implicit parameters are more powerful, in that the binding they create can be set from any reference site. But they do not have explicit crosscutting modularity support because references to the parameter are still spread throughout the code. Many implementations of Scheme provide the fluid-let construct that dynamically binds variables by side-effect, and then re-instates the previous binding is after evaluation of the body is completed.

2.7 Summary

AspectJ is a simple and practical aspect-oriented extension to Java. Programming with AspectJ is a small extension to programming with Java. In AspectJ programs we use the OO constructs to do what they do well, and then use the aspect-oriented constructs to handle the concerns that OO alone cannot effectively modularize.

AspectJ uses a dynamic join point model, in which join points are principled points in the execution of the program. Join points are identified by pointcut designators, and three kinds of advice can be attached to pointcut designators. The simplest pointcut designators identify join points by explicit signature. More powerful pointcut designators identify join points using properties such as whether a method is public or whether a call happens in a particular dynamic context.

Crosscutting concerns implemented in AspectJ are more modular and concise than when written in plain Java. AspectJ also captures the structure of crosscutting concerns in explicit form. Together, these properties make the AspectJ implementation of crosscutting concerns clear and easy to understand.

⁸ Flavors, New Flavors and CLOS use the Common Lisp module system, called the package system. It is typically used in only very coarse-grained ways, certainly not at the level of single classes as in Java, and usually not even at the level of single packages in Java.

Because AspectJ makes the structure of crosscutting concerns more explicit, programming environment tools can help the programmer navigate and understand that structure. Extensions to several popular development tools show that this support is useful when writing AspectJ code.

The main elements of the language design are now fairly stable, but the AspectJ project is not nearly finished. We continue polishing the language design and improving the compiler and tool support. Our goal is to develop and support a large user community. We will work with that community to do empirical studies of the software engineering benefits of AspectJ and AOP.

2.8 Acknowledgements

We thank the AspectJ users most of all. Their questions, comments and bug reports have been invaluable in getting the project to where it is today. Without the users, this project would not be possible.

Brian de Alwis, Yvonne Coady, Chris Dutchyn and Gail Murphy helped us with detailed comments on the paper.

Our work builds on contributions from numerous past members of our research group. In particular John Lamping and Cristina Lopes played major roles in getting AOP and AspectJ to where they are today.

This work was partially supported by the Defense Advanced Research Projects Agency under contract number F30602-C-0246. This work was also partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and Sierra Systems. Java and Forte are trademarks of Sun Microsystems. JBuilder is a trademark of Inprise Corporation.

2.9 References

1. *Proceedings of the Workshop on Domain-Specific Languages (DSL, in association with POPL '97)*. ACM. January 18, 1997.
2. *Proceedings of the Conference on Domain-Specific Languages (DSL)*. USENIX. October 15-17, 1997.
3. *Proceedings of the Conference on Domain-Specific Languages (DSL)*. USENIX. October 3-5, 1999.
4. Aksit, M., L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 615 p. 372-395. Springer. June 29 - July 3, 1992.
5. Aksit, M. and B. Tekinerdogan. Aspect-Oriented Programming Using Composition-Filters. *Position paper for the Aspect Oriented Programming Workshop at the European Conference on Object-Oriented Programming*

- (ECOOP). J.B. Serge Demeyer, Editor. Vol. 1543. Lecture Notes in Computer Science p. 435. Springer. July 20-24, 1998.
6. Bobrow, D.G., et al., Common Lisp Object System Specification, in *Common Lisp: The Language*, 2nd ed. ed. Digital Press. p. 770-864. 1990.
 7. Bobrow, D.G., et al. CommonLoops: Merging Lisp and Object-Oriented Programming. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. N.K. Meyrowitz, Editor. Vol. 11. SIGPLAN Notices p. 17-29. ACM. November 1986.
 8. Cannon, H., *Flavors: A non-hierarchical approach to object-oriented programming*. Symbolics Inc. 1982.
 9. Clifton, C., et al. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM. October 15-19, 2000.
 10. Coady, Y., G. Kiczales, and M. Feeley. Exploring an Aspect-Oriented Approach to Operating System Code. *Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM. October 15-19, 2000.
 11. DeVolder, K. Aspect-Oriented Logic Meta Programming. *Meta-Level Architectures and Reflection, Reflection'99*. P. Cointe, Editor. Vol. 1616 p. 250-272. Springer. 1999.
 12. Filman, R.E. and D.P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. *Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM. October 15-19, 2000.
 13. Friendly, L. Design of Javadoc. *The Design of Distributed Hyperlinked Programming Documentation (IWHD)*. Springer-Verlag. June 1-2, 1995.
 14. Goldberg, A., *Smalltalk-80: The Interactive Programming environment*, Reading MA: Addison-Wesley. 1984.
 15. Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation*: Addison-Wesley. 1983.
 16. Green, T.R.G. and M. Petre, *Usability analysis of visual programming environments: a 'cognitive dimensions' approach*. *Journal of Visual Languages and Computing*, 7(2): p. 131-174. 1996.
 17. Griswold, D., *The Java HotSpot Virtual Machine Architecture*. Sun Microsystems, Inc. 1998.
 18. Ichisugi, Y., S. Matsuoka, and A. Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. *International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*. p. 24-35. November 1992.

19. Irwin, J., et al. Aspect-Oriented Programming of Sparse Matrix Code. *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*. R.R.O. Yutaka Ishikawa, John V. W. Reynders, Marydell Tholburn, Editor. Vol. 1343. Lecture Notes in Computer Science p. 249-256. Springer. December 8-11, 1997.
20. Keene, S.E., *Object-Oriented Programming in Common Lisp*, Reading, Mass: Addison-Wesley Publishing Co. 288. 1989.
21. Kiczales, G., et al. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. LNCS 1241. Springer-Verlag. June 1997.
22. Kiczales, G. and L. Rodriguez. Efficient Method Dispatch in PCL. *LISP and Functional Programming*. p. 99-105. ACM Press. June 27-29, 1990.
23. Lewis, J., et al. Implicit Parameters: Dynamic Scoping with Static Types. *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. p. 108-118. January 2000.
24. Lieberherr, K.J., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, Boston: PWS Publishing Company. 1996.
25. Lopes, C.V., *D: A Language Framework for Distributed Programming*. PhD Thesis. Palo Alto, CA: Northeastern University. 1997.
26. Lopes, C.V. and G. Kiczales, *D: A Language Framework for Distributed Programming*. Xerox Palo Alto Research Center: Palo Alto, CA. SPL97-010, P9710047. 1997.
27. Maes, P. Concepts and Experiments in Computational Reflection. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. N.K. Meyrowitz, Editor. Vol. 12. SIGPLAN Notices p. 147-155. ACM. October 4-8, 1987.
28. Masuhara, H., S. Matsuoka, and A. Yonezawa. Designing an OO reflective language for massively-parallel processors. *Position paper for the workshop on Object-Oriented Reflection and Metalevel Architectures at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 26 September - 1 October, 1993.
29. Matsuoka, S., T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 512 p. 231-250. Springer. July 15-19, 1991.
30. McAffer, J. The CodA MOP. *Position paper for the workshop on Object-Oriented Reflection and Metalevel Architectures at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*,. 26 September - 1 October.

31. Mendhekar, A., G. Kiczales, and J. Lamping, *RG: A Case-Study for Aspect-Oriented Programming*. Xerox Palo Alto Research Center: Palo Alto, CA. SPL97-009, P9710044. 1997.
32. Mezini, M. and K.J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 33. SIGPLAN Notices p. 97-116. ACM. October 18-22, 1998.
33. Moon, D.A. Object-Oriented Programming with Flavors. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. N.K. Meyrowitz, Editor. Vol. 11. SIGPLAN Notices p. 1-8. ACM. November 1986.
34. Okamura, H., Y. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. *International Symposium on Object Technologies for Advanced Software*. p. 110-127. Springer Verlag. 1993.
35. Ossher, H., et al. Subject-Oriented Composition Rules. *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Vol. 30. SIGPLAN Notices p. 235-250. ACM. October 15-19 1995.
36. Ossher, H. and P.L. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. *Proceedings of the International Conference on Software Engineering (ICSE)*. p. 734-737. ACM. June 4-11, 2000.
37. Parnas, D.L., *On the Criteria To Be Used in Decomposing Systems Into Modules*. Communications of the ACM, 15(12): p. 1053-1058. 1972.
38. Parnas, D.L., *Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs*. Lecture Notes in Computer Science, Programming Methodology. 1974.
39. Shneiderman, B., Direct Manipulation: A step beyond Programming languages, in *Human-Computer Interaction: A Multidisciplinary Approach*. R.M. Baecker and W.A.S. Buxton, Editors. Morgan Kaufmann Publishers, Inc.: Los Altos, CA. p. 461-467. 1983.
40. Smith, B.C., *Reflection and Semantics in a Procedural Language*. PhD: M.I.T. 1982.
41. Smith, B.C. Reflection and Semantics in LISP. *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. Vol. 11 p. 23-35. ACM. 1983.
42. Steele, G.L., *Common Lisp the Language*. 2nd ed: Digital Press. 1029. 1990.
43. Stroustrup, B., *The C++ Programming Language*. 3rd ed: Addison-Wesley. 1997.
44. Tarr, P.L., et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the International Conference on Software Engineering (ICSE)*. p. 107-119. ACM. May 16-22, 1999.

45. Walker, R. and G. Murphy. Implicit Context: Easing Software Evolution and Reuse. *Proceedings of the Conference on Foundations of Software Engineering (FSE)*. ACM. November 6-10, 2000.
46. Watanabe, T. and A. Yonezawa. Reflection in an object-oriented concurrent language. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. p. 306-315. ACM. 1988.

3 Getting Started with AspectJ

Many software developers are attracted to the idea of AOP, but unsure about how to begin using the technology. They recognize the concept of crosscutting concerns, and know that they have had problems with the implementation of such concerns in the past. But there are many questions about how to adopt AOP into the development process. Common questions include: Can I use aspects in my existing code? What kinds of benefits can I expect to get? How do I find aspects? How steep is the learning curve for AOP? What are the risks of using this new technology?

This paper addresses these questions in the context of AspectJ – a general-purpose aspect-oriented extension to Java. A series of abridged examples illustrate the kinds of aspects programmers may want to implement using AspectJ and the benefits associated with doing so. Readers who would like to understand the examples in more detail, or who want to learn how to program examples like these, can find the complete examples and supporting material on the AspectJ web site.⁹

A significant risk in adopting any new technology is going too far too fast. Concern about this risk causes many organizations to be conservative about adopting new technology. To address this issue, the examples in the paper are grouped into three broad categories, with aspects that are easier to adopt into existing development projects coming earlier in the paper. Section 3.2 presents *development aspects* that facilitate tasks such as debugging, testing and performance tuning of applications. Section 3.3 presents *production aspects* that implement crosscutting functionality common in Java applications.

These categories are informal, and this ordering is not the only way to adopt AspectJ. Some developers may want to use a production aspect right away. But our experience with current AspectJ users suggests that this is one ordering that allows developers to get experience with (and benefit from) AOP technology quickly, while also minimizing risk.

⁹ The AspectJ system, primer, and a complete implementation of these examples are available at <http://aspectj.org>. (Note to reviewers: We will make a special place on the web site to get these examples. For a few months after the CACM issue comes out this will be prominently featured. Later it will be less prominently featured, but will remain on the site indefinitely.)

3.1 AspectJ semantics

This section presents a brief introduction to the features of AspectJ used later in the paper. These features are at the core of the language, but this is by no means a complete overview of AspectJ. For a more complete or more detailed understanding of AspectJ, see [2, 3].

The semantics are presented using a simple figure editor system. A *Figure* consists of a number of *FigureElements*, which can be either *Points* or *Lines*. The *Figure* class provides factory services. There is also a *Display*. Most example programs later in the paper are based on this system as well.

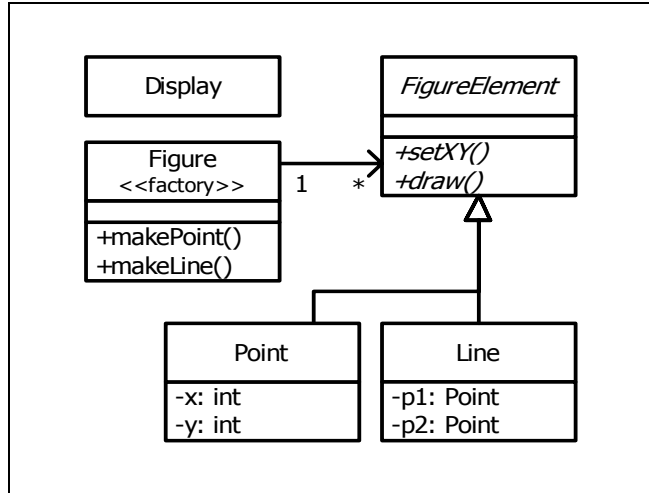


Figure 7 UML for Figure Editor Example.

AspectJ realizes the modularization of crosscutting concerns using join points and advice. *Join points* are well-defined points in the program flow and *advice* define code that is executed when join points are reached.

3.1.1 The Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the structure of crosscutting concerns.

In AspectJ, join points are certain well-defined points in the execution of the program. AspectJ provides for many kinds of join points, but this paper discusses only one of them: method call join points. A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including normal or abrupt return.

Each method call itself is one join point. The dynamic context of a method call may include many other join points: all the join points that occur when executing the called method and any methods that it calls.

3.1.2 Pointcut Designators

In AspectJ, *pointcut designators* identify collections of join points in the program flow. For example, the pointcut designator:

```
calls(void Point.setX(int))
```

identifies all calls to the method `setX` defined on `Point` objects. Pointcut designators can be composed using a set algebra semantics, so for example:

```
calls(void Point.setX(int)) ||
calls(void Point.setY(int))
```

identifies all calls to either the `setX` or `setY` methods defined by `Point`.

Programmers can define their own pointcut designators, and pointcut designators can identify join points from many different classes – in other words, they can crosscut classes. So, for example, the following named pointcut declaration

```
pointcut moves():
    calls(void FigureElement.setXY(int, int)) ||
    calls(void Point.setX(int))                ||
    calls(void Point.setY(int))                ||
    calls(void Line.setP1(Point))              ||
    calls(void Line.setP2(Point));
```

defines a pointcut named `moves` that designates calls to any of the methods that move figure elements.

3.1.2.1 Property-Based Primitive Pointcut Designators

The previous pointcut designators are all based on explicit enumeration of a set of method signatures. We call this *name-based* crosscutting. AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. We call this *property-based* crosscutting. The simplest of these involve using wildcards in certain fields of the method signature. For example:

```
calls(void Figure.make*(..))
```

identifies calls to any method defined on `Figure`, for which the name begins with "make", specifically the factory methods `makePoint` and `makeLine`; and

```
calls(public * Figure.* (..)) ||
```

identifies calls to any public method defined on `Figure`.

One very powerful primitive pointcut designator, `cflow`, identifies join points based on whether they occur in the dynamic context of another pointcut. So

```
cflow(moves())
```

designates all join points that occur between receiving method calls for the methods in `moves` and returning from those calls (either normally or by throwing a `Throwable`).

3.1.3 Advice

Pointcuts are used in the definition of advice. AspectJ has several different kinds of advice that define additional code that should run at join points. *Before advice* runs when a join point is reached and before the computation proceeds, i.e. that runs when computation reaches the method call and before the actual method starts running. *After advice* runs after the computation 'under the join point' finishes, i.e. after the method body has run, and just before control is returned to the caller. *Around advice* runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all.


```

after(): moves() {
    System.out.println("A figure element moved.");
}

```

3.1.3.1 Exposing Context in Pointcuts

Pointcut designators can also expose part of the execution context at their join points. Values exposed by a pointcut designator can be used in the body of advice declarations. In the following code, the pointcut exposes three values from calls to `setXY`: the `FigureElement` receiving the call, the new value for `x` and the new value for `y`. The advice then prints the figure element that was moved and its new `x` and `y` coordinates after each `setXY` method call.

```

pointcut setXYs(FigureElement fe, int x, int y):
    calls(void fe.setXY(x, y));

after(FigureElement fe, int x, int y): setXYs(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}

```

3.1.4 Aspect Declarations

An *aspect* is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers. The crosscutting implementation is provided in terms of pointcuts and advice. Only aspects may include advice, so while AspectJ may define crosscutting effects, the declaration of those effects is localized.

3.2 Development Aspects

This section presents examples of aspects that can be used during development of Java applications. These aspects facilitate debugging, testing and performance tuning work. The aspects define behavior that ranges from simple tracing, to profiling, to testing of internal consistency within the application.

Using AspectJ makes it possible to cleanly modularize this kind of functionality, thereby making it possible to easily enable and disable the functionality when desired. Section 3.2.4 presents techniques that make it possible to ensure that the functionality is not included in production builds of an application. These techniques give developers who have reason to be conservative about new technology adoption a strong intermediate position from which to start using AspectJ. They can use AspectJ for debugging and some testing, but still compile and ship the production code without aspects.

3.2.1 Tracing, logging, profiling

A first example is a simple tracing aspect that just prints a simple message at the specified method calls. Continuing with the figure editor example, one such aspect might simply trace whenever points are moved.

```

aspect SimpleTracing {
    pointcut tracedCalls():
        calls(void FigureElement.draw(GraphicsContext));

    before(): tracedCalls() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}

```

This code makes use of the `thisJoinPoint` special variable. Within all advice bodies this variable is bound to an object that describes the current join point. The effect of this code is to print a line like the following every time a figure element receives a draw method call:

```

Entering: call(void FigureElement.draw(GraphicsContext))

```

To understand the benefit of coding this with AspectJ consider changing the set of method calls that are traced. With AspectJ, this just requires editing the definition of the `tracedCalls` pointcut and recompiling. The individual methods that are traced do not need to be edited.

When debugging, programmers often invest considerable effort in figuring out a good set of trace points to use when looking for a particular kind of problem. When debugging is complete – or appears to be complete – it is frustrating to have to lose that investment by deleting trace statements from the code. The alternative of just commenting them out makes the code look bad, and can cause trace statements for one kind of debugging to get confused with trace statements for another kind of debugging.

With AspectJ it is easy to both preserve the work of designing a good set of trace points and disable the tracing when it isn't being used. This is done by writing an aspect specifically for that tracing mode, and removing that aspect from the compilation when it is not needed.

This ability to concisely implement and reuse debugging configurations that have proven useful in the past is a direct result of AspectJ modularizing a crosscutting design element – the set of methods that are appropriate to trace when looking for a given kind of information.

3.2.1.1 Profiling and Logging

There are many sophisticated profiling tools available on the market. These can gather a variety of information and display the results in useful ways. But sometimes programmers want very specific profiling or logging behavior. In these cases it is often possible to write a simple aspect similar to the ones above to do the job.

For example, the following aspect will count the number of calls to the `rotate` method on a `Line` and the number of calls to the `set*` methods of a `Point` that happen within the control flow of those calls to `rotate`:

```

aspect SetsInRotateCounting {
    int rotateCount = 0;
    int setCount    = 0;

    before(): calls(void Line.rotate(double)) {
        rotateCount++;
    }
    before(): calls(void Point.set*(int)) &&
        cflow(calls(void Line.rotate(double))) {
        setCount++;
    }
}

```

3.2.2 Pre/post conditions

Many programmers use the “Design by Contract” style popularized by Eiffel [1]. In this style of programming, explicit pre-conditions test that callers of a method call it properly and explicit post-conditions test that methods properly do the work they are supposed to.

AspectJ makes it possible to implement pre- and post-condition testing in modular form. For example, this code

```

aspect PointBoundsChecking {
    pointcut setXs(int x):
        calls(void FigureElement.setXY(x, int)) ||
        calls(void Point.setX(x));

    pointcut setYs(int y): ...;

    before(int x): setXs(x) {
        if ( x < MIN_X || x > MAX_X ) {
            throw new IllegalArgumentException ("x is out of bounds.");
        }
    }

    before(int y): setYs(y) {
        ...
    }
}

```

implements the bounds checking aspect of pre-condition testing for operations that move points. Notice that the `setXs` pointcut designator refers to all the operations that can set a point’s `x` coordinate; this includes the `setX` method, as well as “half of” the `setXY` method. In this sense the `setXs` pointcut can be seen as involving very fine-grained crosscutting – it names the `setX` method and half of the `setXY` method.

Even though pre- and post-condition testing aspects can often be used only during testing, in some cases developers may wish to include them in the production build as well. Again, because AspectJ makes it possible to cleanly modularize these crosscutting concerns, it gives developers good control over this decision.

3.2.3 Contract enforcement

The property-based crosscutting mechanisms can be very useful in defining more sophisticated contract enforcement. One very powerful use of these mechanisms is to identify method calls that, in a correct program, should not exist. For example, the following aspect enforces the constraint that only the well-known factory methods can add an element to the registry of figure elements. Enforcing this constraint ensures that no figure element is added to the registry more than once.

```
static aspect RegistrationProtection {
    pointcut registers():
        calls(void Registry.register(FigureElement));

    pointcut canRegister():
        withincode(static * FigureElement.make*(..));

    before(): registers() && !canRegisters() {
        throw new IllegalArgumentException("Illegal call " +
thisJoinPoint);
    }
}
```

This aspect uses the `withincode` primitive pointcut designator to denote all join points that occur within the body of the factory methods on `FigureElement` (the methods with names that begin with "make"). This is a property-based pointcut designator because it identifies join points based not on their signature, but rather on the property that they occur specifically within the code of another method. The `before` advice declaration effectively says “signal an error for any calls to `register` that are not within the factory methods.”

3.2.4 Configuration Management

Configuration management for aspects can be handled using a variety of “make-file like” techniques. To work with optional aspects, the programmer can simply define their make files to either include the aspect in the call to the AspectJ compiler or not, as desired.

Developers who want to be certain that no aspects are included in the production build can do so by configuring their make files so that they use a traditional Java compiler for production builds. To make it easy to write such make files, the AspectJ compiler has a command-line interface that is consistent with ordinary Java compilers.

3.3 Production Aspects

This section presents examples of aspects that are inherently intended to be included in production builds of an application. Again, we begin with named-based aspects and follow with property-based aspects. Name-based production aspects tend to affect only a small number of methods. For this reason, they are a good next step for projects adopting AspectJ. But even though they tend to be small and simple, they can often have a significant effect in terms of making the program easier to understand and maintain.

3.3.1 Change Monitoring

The first example production aspect supports the code that refreshes the display. The role of the aspect is to maintain a dirty bit indicating whether or not an object has moved since the last time the display was refreshed.

Implementing this functionality as an aspect is straightforward. The `testAndClear` method is called by the display code to find out whether a figure element has moved recently. This method returns the current state of the `dirty` flag and resets it to `false`. The `moves` pointcut captures all the method calls that can move a figure element. The `after` advice on `moves` sets the `dirty` flag whenever an object moves.

```
aspect MoveTracking {
    private static boolean dirty = false;

    public static synchronized boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;
    }

    pointcut moves():
        calls(void FigureElement.setXY(int, int)) ||
        calls(void Line.setP1(Point)) ||
        calls(void Line.setP2(Point)) ||
        calls(void Point.setX(int)) ||
        calls(void Point.setY(int));

    after(): moves() {
        dirty = true;
    }
}
```

Even this simple example serves to illustrate some of the important benefits of using AspectJ in production code. Consider implementing this functionality with ordinary Java: there would likely be a helper class that contained the `dirty` flag, the `testAndClear` method as well as a `setFlag` method. Each of the methods that could move a figure element would include a call to the `setFlag` method. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case.

The AspectJ implementation has several advantages over the standard implementation:

The structure of the crosscutting concern is captured explicitly. The `moves` pointcut clearly states all the methods involved, so the programmer reading the code sees not just individual calls to `setFlag`, but instead sees the real structure of the code. As shown in Figure 8 the IDE support included with AspectJ, will automatically remind the programmer that this aspect advises each of the methods involved.

Evolution is easier. If, for example, the aspect needs to be revised to record not just that some figure element moved, but rather to record exactly which figure elements moved, the change would be entirely local to the aspect. The pointcut would be updated to expose the object being moved, and the advice would be updated to record that object. ([3] presents a detailed discussion of various ways this aspect could be expected evolve.)

The functionality is easy to plug in and out. Just as with development aspects, production aspects may need to be removed from the system, either because the functionality is no longer needed at all, or because it is not needed in certain configurations of a system. Because the functionality is modularized in a single aspect this is easy to do.

The implementation is more stable. If, for example, the programmer adds a subclass of `Line` that overrides the existing methods, this advice in this aspect will still apply. In the ordinary Java implementation the programmer would have to remember to add the call to `setFlag` in the new overriding method. This benefit is often even more compelling for property-based aspects (see Section 3.3.4).

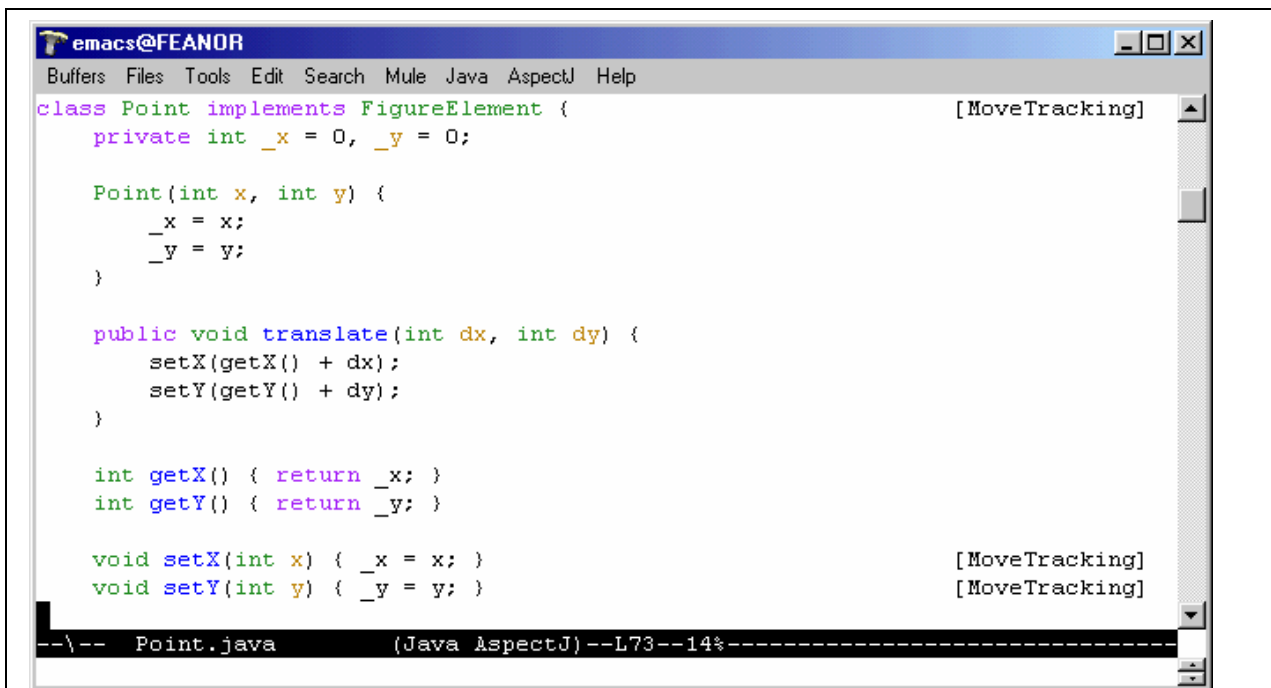


Figure 8 A snapshot of screen when using the AspectJ-aware extension to emacs.

The text in [Square Brackets] following the method declarations is automatically generated, and serves to remind the programmer of the aspects that crosscut the method. The editor also provide commands to jump to the advice from the method and vice-versa.

3.3.2 Synchronization

Another good use of name-based production aspects is to implement synchronization policies. These aspects are similar to change monitoring, except that the work done by the advice tends to be more complex, and these aspects usually use paired before and after advice to handle the synchronization work.

The following example shows how the readers and writers synchronization pattern presented in [4] can be implemented using AspectJ. This aspect uses the `eachobject` feature of the language to ensure that each object to which this aspect applies will have its own instance of the aspect, and therefore its own count of active and waiting readers and writers. This means that the synchronization constraints of this aspect will apply on a per-object basis, which is appropriate for this pattern.

```
aspect RegistryReaderWriterSynchronizing
  of eachobject(instanceof(readers() || writers())) {
  pointcut readers():
    calls(Vector Registry.elementsNear(int, int));

  pointcut writers():
    calls(void Registry.add(FigureElement)) ||
    calls(void Registry.remove(FigureElement));

  protected int activeReaders, activeWriters,
    waitingReaders, waitingWriters;

  before(): readers() { beforeRead(); } //these helper
  after(): readers() { afterRead(); } //methods of the
  before(): writers() { beforeWrite(); } //aspect are
  after(): writers() { afterWrite(); } //defined below

  protected synchronized void beforeRead() {
    ++waitingReaders;
    while (!(waitingWriters == 0 && activeWriters == 0)) {
      try { wait(); } catch (InterruptedException ex) {}
    }
    --waitingReaders;
    ++activeReaders;
  }

  protected synchronized void afterRead() { ... }
  protected synchronized void beforeWrite() { ... }
  protected synchronized void afterWrite() { ... }
}
```

3.3.3 Context Passing

The crosscutting structure of context passing can be a significant source of complexity in Java programs. Consider implementing functionality that would allow a client of the figure editor (a program client rather than a human) to set the color of any figure elements that are created. Typically this requires passing a color, or a color factory, from

the client, down through the calls that lead to the figure element factory. All programmers are familiar with the inconvenience of adding a first argument to a number of methods just to pass this kind of context information.

Using AspectJ, this kind of context passing can be implemented in a modular way. The following code adds after advice that runs only when the factory methods of `Figure` are called in the control flow of a method on a `ColorControllingClient`.

```
aspect ColorControl {
    pointcut CCClientCflow(ColorControllingClient client):
        cflow(calls(* client.* (..)));

    pointcut makes(FigureElement fe):
        calls(fe Figure.make*(..));

    after (ColorControllingClient c, FigureElement fe):
        makes(fe) && CCClientCflow(c) {
        fe.setColor(c.colorFor(e));
    }
}
```

This aspect affects only a small number of methods, but note that the non-AOP implementation of this functionality might require editing many more methods, specifically, all the methods in the control flow from the client to the factory. This is a benefit common to many property-based aspects – while the aspect is short and affects only a modest number of benefits, the complexity the aspect saves is potentially much larger.

3.3.4 Consistent Behavior Across a Large Number of Operations

This example aspect shows how a property-based aspect can be used to provide consistent handling of functionality across a large set of operations. This aspect ensures that all public methods of the `com.xerox` package log any errors they throw to their caller. The `publicCalls` pointcut captures the public method calls of the package, and the after advice runs whenever one of those calls returns throwing an exception. The advice logs the exception and then the throw resumes.

```
aspect PublicErrorLogging {
    Log log = new Log();

    pointcut publicMethodCalls ():
        calls(public * com.xerox.*.*(..));

    after() throwing (Error e): publicMethodCalls() {
        log.write(e);
    }
}
```

In some cases this aspect can log an exception twice. This happens if code inside the `com.xerox` package itself calls a public method of the package. In that case this code will log the error at both the outermost call into the `com.xerox` package and the re-entrant call. The `cflow` primitive pointcut can be used in a nice way to exclude these re-entrant calls:


```

after() throwing (Error e): publicMethodCalls() &&
                                !cflow(publicMethodCalls()) {
    log.write(e);
}

```

The following aspect is taken from work on the AspectJ compiler. The aspect advises about 35 methods in the `JavaParser` class. The individual methods handle each of the different kinds of elements that must be parsed. They have names like `parseMethodDec`, `parseThrows`, and `parseExpr`.

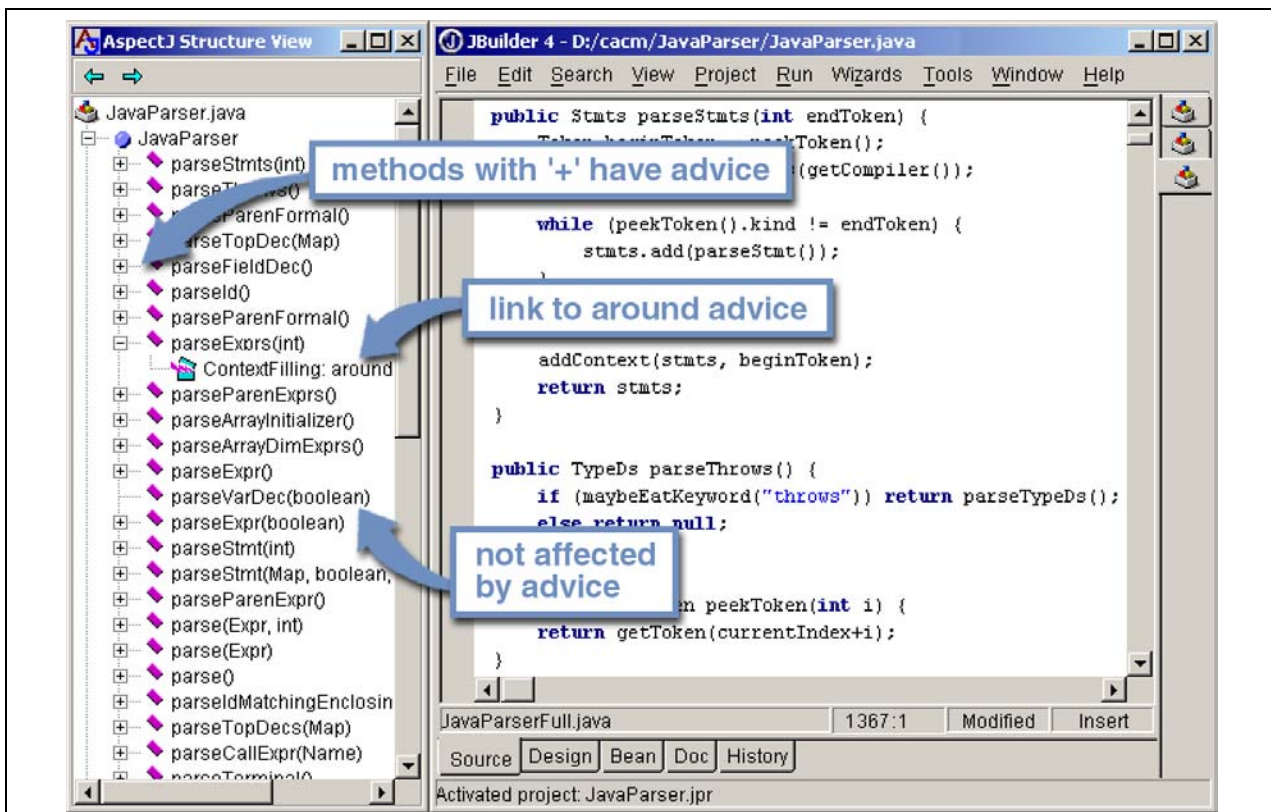


Figure 9 Examining the partially re-factored code with the AJDE extension to JBuilder.

The method names marked with lightning? Are advised by the aspect. This view allows us to quickly see the structure of this larger aspect. Being able to see this perspective of the aspect is very helpful to designing and debugging property-based aspects like this one.

```

aspect ContextFilling {
    pointcut parses(JavaParser jp):
        calls(* jp.parse*(..) &&
             !calls(Stmt parseVarDec(boolean))); // var decs
                                                // are tricky

    around(JavaParser jp) returns ASTObject: parses(jp) {
        Token beginToken = jp.peekToken();
        ASTObject ret = proceed(jp);
        if (ret != null) jp.addContext(ret, beginToken);
        return ret;
    }
}

```

This example exhibits a property found in many aspects with large property-based pointcuts. In addition to a general property based pattern – `calls(* jp.parse*(..))` – it includes an exception to the pattern – `!calls(Stmt parseVarDec(boolean))`. The exclusion of `parseVarDec` happens because the parsing of variable declarations in Java is too complex to fit with the clean pattern of the other `parse*` methods. Even with the explicit exclusion this aspect is a clear expression of a clean crosscutting modularity. Namely that all `parse*` methods that return `ASTObjects`, except for `parseVarDec` share a common behavior for establishing the parse context of their result.

The process of writing an aspect with a large property-based pointcut, and of developing the appropriate exceptions can clarify the structure of the system. This is especially true, as in this case, when refactoring existing code to use aspects. When we first looked at the code for this aspect, we were able to use the IDE support provided in `AJDEforJBuilder` to see what methods the aspect was advising as compared to where we had manually previously manually coded the functionality. We used the `AJDE` structure view shown in Figure 9 and scrolled through the code. We quickly discovered that there were a dozen places where the aspect advice was in effect but we had not written the manual coding of the context functionality. Two of these were bugs in our prior non-AOP implementation of the parser. The other ten were needless performance optimizations. So in this case refactoring the code to express the crosscutting structure of the aspect explicitly made the code more concise and eliminated latent bugs.

3.4 Conclusion

AspectJ is a simple and practical aspect-oriented extension to Java™. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns.

Adoption of AspectJ into an existing Java development project can be a straightforward task and incremental task. One path is to begin by using only development aspects, going on to using production aspects after building up experience with AspectJ. Adoption can follow other paths as well. For example, some developers will benefit from using production aspects right away.

AspectJ enables both name-based and property based crosscutting. Aspects that use name-based crosscutting tend to affect a small number of other classes. But despite their

small scale, they can often eliminate significant complexity compared to an ordinary Java implementation. Aspects that use property-based crosscutting can have small or large scale.

Using AspectJ results in clean well-modularized implementations of crosscutting concerns. When written as an AspectJ aspect the structure of a crosscutting concern is explicit and easy to understand. Aspects are also highly modular, making it possible to develop plug-and-play implementations of crosscutting functionality.

AspectJ provides more functionality than is covered by this short article, but these examples should provide a sense of the kinds of aspects it is possible to write using AspectJ. But we recommend that programmers read the on-line AspectJ documentation and examples carefully before deciding to adopt AspectJ into a project.

3.5 Acknowledgements

We thank the AspectJ users most of all.

Tom Roeder contributed the initial implementation of parser context aspect. Yvonne Coady, Kris De Volder, Chris Dutchyn, Jan Hanneman and Gail Murphy made extensive comments on early drafts of the paper.

3.6 References

1. *Bug-Free O-O Software: An Introduction to Design by Contract*, in *Web page at <http://eiffel.com/doc/manuals/technology/contract/page.html>*. 1999, Interactive Software Engineering.
2. *The AspectJ Primer*, in *Web page at <http://aspectj.org/docs/primer>*. 2001, The AspectJ Team.
3. Kiczales, G., et al. An Overview of AspectJ. *15th European Conference on Object Oriented Programming (ECOOP)*. Springer. June 2001.
4. Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*. 2nd Edition ed: Addison-Wesley. 1999.

4 The AspectJ™ Programming Guide

4.1 Preface

This programming guide does three things. It introduces the AspectJ language defines each of AspectJ's constructs and their semantics, and provides examples of their use.

It includes appendices that give a reference to the syntax of AspectJ, a more formal description of AspectJ's semantics, and a description of limitations allowed by AspectJ implementations.

The first section, [Getting Started with AspectJ](#), provides a gentle overview of writing AspectJ programs. It also shows how one can introduce AspectJ into an existing development effort in stages, reducing the associated risk. You should read this section if this is your first exposure to AspectJ and you want to get a sense of what AspectJ is all about.

The second section, [The AspectJ Language](#), covers the features of the language in more detail, using code snippets as examples. All the basics of the language is covered, and after reading this section, you should be able to use the language correctly.

The next section, [Examples](#), comprises a set of complete programs that not only show the features being used, but also try to illustrate recommended practice. You should read this section after you are familiar with the elements of AspectJ.

Finally, there are two short chapters, one on [Idioms](#) and one on [Pitfalls](#).

The back matter contains several appendices that cover a [AspectJ Quick Reference](#) to the language's syntax, a more in depth coverage of its [Semantics](#), and a description of the latitude enjoyed by its [Implementation Limitations](#).

4.2 Getting Started with AspectJ

Many software developers are attracted to the idea of aspect-oriented programming (AOP) but unsure about how to begin using the technology. They recognize the concept of crosscutting concerns, and know that they have had problems with the implementation of such concerns in the past. But there are many questions about how to adopt AOP into the development process. Common questions include:

Can I use aspects in my existing code?

What kinds of benefits can I expect to get from using aspects?

How do I find aspects in my programs?

How steep is the learning curve for AOP?

What are the risks of using this new technology?

This chapter addresses these questions in the context of AspectJ: a general-purpose aspect-oriented extension to Java. A series of abridged examples illustrate the kinds of aspects programmers may want to implement using AspectJ and the benefits associated with doing so. Readers who would like to understand the examples in more detail, or who want to learn how to program examples like these, can find more complete examples and supporting material linked from the AspectJ web site (<http://eclipse.org/aspectj>).

A significant risk in adopting any new technology is going too far too fast. Concern about this risk causes many organizations to be conservative about adopting new technology. To address this issue, the examples in this chapter are grouped into three broad categories, with aspects that are easier to adopt into existing development projects coming earlier in this chapter. The next section, [Introduction to AspectJ](#), we present the core of AspectJ's features, and in [Development Aspects](#), we present aspects that facilitate tasks such as debugging, testing and performance tuning of applications. And, in the section following, [Production Aspects](#), we present aspects that implement crosscutting functionality common in Java applications. We will defer discussing a third category of aspects, reusable aspects, until [The AspectJ Language](#).

These categories are informal, and this ordering is not the only way to adopt AspectJ. Some developers may want to use a production aspect right away. But our experience with current AspectJ users suggests that this is one ordering that allows developers to get experience with (and benefit from) AOP technology quickly, while also minimizing risk.

4.2.1 Introduction to AspectJ

This section presents a brief introduction to the features of AspectJ used later in this chapter. These features are at the core of the language, but this is by no means a complete overview of AspectJ.

The features are presented using a simple figure editor system. A `Figure` consists of a number of `FigureElements`, which can be either `Points` or `Lines`. The `Figure` class provides factory services. There is also a `Display`. Most example programs later in this chapter are based on this system as well.

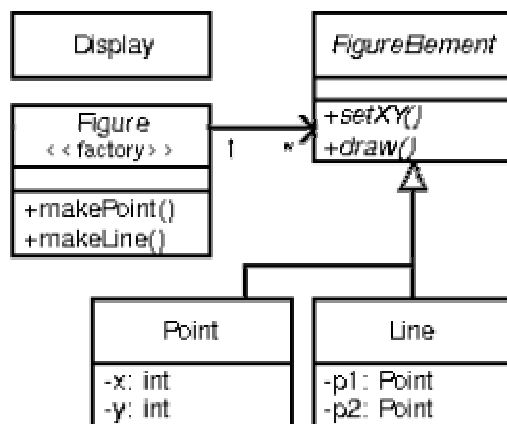


Figure 10 UML for the FigureEditor example

The motivation for AspectJ (and likewise for aspect-oriented programming) is the realization that there are issues or concerns that are not well captured by traditional programming methodologies. Consider the problem of enforcing a security policy in some application. By its nature, security cuts across many of the natural units of modularity of the application. Moreover, the security policy must be uniformly applied to any additions as the application evolves. And the security policy that is being applied might itself evolve. Capturing concerns like a security policy in a disciplined way is difficult and error-prone in a traditional programming language.

Concerns like security cut across the natural units of modularity. For object-oriented programming languages, the natural unit of modularity is the class. But in object-oriented programming languages, crosscutting concerns are not easily turned into classes precisely because they cut across classes, and so these aren't reusable, they can't be refined or inherited, they are spread through out the program in an undisciplined way, in short, they are difficult to work with.

Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns. AspectJ is an implementation of aspect-oriented programming for Java.

AspectJ adds to Java just one new concept, a join point -- and that's really just a name for an existing Java concept. It adds to Java only a few new constructs: pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, inter-type declarations statically affects a program's class heirarchy, and aspects encapsulate these new constructs.

A *join point* is a well-defined point in the program flow. A *pointcut* picks out certain join points and values at those points. A piece of *advice* is code that is executed when a join point is reached. These are the dynamic parts of AspectJ.

AspectJ also has different kinds of *inter-type declarations* that allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes.

AspectJ's *aspect* are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations.

In the sections immediately following, we are first going to look at join points and how they compose into pointcuts. Then we will look at advice, the code which is run when a pointcut is reached. We will see how to combine pointcuts and advice into aspects, AspectJ's reusable, inheritable unit of modularity. Lastly, we will look at how to use inter-type declarations to deal with crosscutting concerns of a program's class structure.

4.2.1.1 The Dynamic Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns. This chapter describes AspectJ's dynamic join points, in which join points are certain well-defined points in the execution of the program.

AspectJ provides for many kinds of join points, but this chapter discusses only one of them: method call join points. A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception).

Each method call at runtime is a different join point, even if it comes from the same call expression in the program. Many other join points may run while a method call join point is executing -- all the join points that happen while executing the method body, and in those methods called from the body. We say that these join points execute in the *dynamic context* of the original call join point.

4.2.1.2 Pointcuts

In AspectJ, *pointcuts* pick out certain join points in the program flow. For example, the pointcut

```
call(void Point.setX(int))
```

picks out each join point that is a call to a method that has the signature `void Point.setX(int)` — that is, `Point`'s `void setX` method with a single `int` parameter.

A pointcut can be built out of other pointcuts with `and`, `or`, and `not` (spelled `&&`, `||`, and `!`). For example:

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

picks out each join point that is either a call to `setX` or a call to `setY`.

Pointcuts can identify join points from many different types — in other words, they can crosscut types. For example,

```
call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int))                ||
call(void Point.setY(int))                ||
call(void Line.setP1(Point))              ||
call(void Line.setP2(Point));
```

picks out each join point that is a call to one of five methods (the first of which is an interface method, by the way).

In our example system, this pointcut captures all the join points when a `FigureElement` moves. While this is a useful way to specify this crosscutting concern, it is a bit of a mouthful. So AspectJ allows programmers to define their own named pointcuts with the `pointcut` form. So the following declares a new, named pointcut:

```
pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))                ||
    call(void Point.setY(int))                ||
    call(void Line.setP1(Point))              ||
    call(void Line.setP2(Point));
```

and whenever this definition is visible, the programmer can simply use `move()` to capture this complicated pointcut.

The previous pointcuts are all based on explicit enumeration of a set of method signatures. We sometimes call this *name-based* crosscutting. AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. We call this *property-based* crosscutting. The simplest of these involve using wildcards in certain fields of the method signature. For example, the pointcut

```
call(void Figure.make*(..))
```

picks out each join point that's a call to a void method defined on `Figure` whose the name begins with "make" regardless of the method's parameters. In our system, this picks out calls to the factory methods `makePoint` and `makeLine`. The pointcut

```
call(public * Figure.* (..))
```

picks out each call to `Figure`'s public methods.

But wildcards aren't the only properties AspectJ supports. Another pointcut, `cflow`, identifies join points based on whether they occur in the dynamic context of other join points. So

```
cflow(move())
```

picks out each join point that occurs in the dynamic context of the join points picked out by `move()`, our named pointcut defined above. So this picks out each join points that occurs between when a `move` method is called and when it returns (either normally or by throwing an exception).

4.2.1.3 Advice

So pointcuts pick out join points. But they don't *do* anything apart from picking out join points. To actually implement crosscutting behavior, we use advice. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points).

AspectJ has several different kinds of advice. *Before advice* runs as a join point is reached, before the program proceeds with the join point. For example, before advice on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated.

```
before(): move() {  
    System.out.println("about to move");  
}
```

After advice on a particular join point runs after the program proceeds with that join point. For example, after advice on a method call join point runs after the method body has run, just before before control is returned to the caller. Because Java programs can leave a join point 'normally' or by throwing an exception, there are three kinds of after advice: *after returning*, *after throwing*, and *plain after* (which runs after returning *or* throwing, like Java's `finally`).


```
after() returning: move() {
    System.out.println("just successfully moved");
}
```

Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point. Around advice is not discussed in this section.

4.2.1.3.1 Exposing Context in Pointcuts

Pointcuts not only pick out join points, they can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations.

An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses. For example, the after advice

```
after(FigureElement fe, int x, int y) returning:
    ...SomePointcut... {
    ...SomeBody...
}
```

uses three pieces of exposed context, a `FigureElement` named `fe`, and two `ints` named `x` and `y`.

The body of the advice uses the names just like method parameters, so

```
after(FigureElement fe, int x, int y) returning:
    ...SomePointcut... {
        System.out.println(fe + " moved to (" + x + ", " + y + ")");
    }
```

The advice's pointcut publishes the values for the advice's arguments. The three primitive pointcuts `this`, `target` and `args` are used to publish these values. So now we can write the complete piece of advice:

```
after(FigureElement fe, int x, int y) returning:
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y) {
        System.out.println(fe + " moved to (" + x + ", " + y + ")");
    }
```

The pointcut exposes three values from calls to `setXY`: the target `FigureElement` -- which it publishes as `fe`, so it becomes the first argument to the after advice -- and the two `int` arguments -- which it publishes as `x` and `y`, so they become the second and third argument to the after advice.

So the advice prints the figure element that was moved and its new `x` and `y` coordinates after each `setXY` method call.

A named pointcut may have parameters like a piece of advice. When the named pointcut is used (by advice, or in another named pointcut), it publishes its context by name just like the `this`, `target` and `args` pointcut. So another way to write the above advice is

```

pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);

after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}

```

4.2.1.4 Inter-type declarations

Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. Unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time.

Consider the problem of expressing a capability shared by some existing classes that are already part of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to *each affected class* a method that implements this interface.

AspectJ can express the concern in one place, by using inter-type declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and associates the methods and fields to the existing classes.

Suppose we want to have `Screen` objects observe changes to `Point` objects, where `Point` is an existing class. We can implement this by writing an aspect declaring that the class `Point` has an instance field, `observers`, that keeps track of the `Screen` objects that are observing `Points`.

```

aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}

```

The `observers` field is private, so only `PointObserving` can see it. So observers are added or removed with the static methods `addObserver` and `removeObserver` on the aspect.

```

aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    ...
}

```

Along with this, we can define a pointcut `changes` that defines what we want to observe, and the after advice defines what we want to do when we observe a change.

```

aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }

    pointcut changes(Point p): target(p) && call(void Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }

    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}

```

Note that neither `Screen`'s nor `Point`'s code has to be modified, and that all the changes needed to support this new capability are local to this aspect.

4.2.1.5 Aspects

Aspects wrap up pointcuts, advice, and inter-type declarations in a a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers in addition to the crosscutting members. Because only aspects may include these crosscutting members, the declaration of these effects is localized.

Like classes, aspects may be instantiated, but AspectJ controls how that instantiation happens -- so you can't use Java's `new` form to build new aspect instances. By default, each aspect is a singleton, so one aspect instance is created. This means that advice may use non-static fields of the aspect, if it needs to keep state around:

```

aspect Logging {
    OutputStream logStream = System.err;

    before(): move() {
        logStream.println("about to move");
    }
}

```

Aspects may also have more complicated rules for instantiation, but these will be described in a later chapter.

4.2.2 Development Aspects

The next two sections present the use of aspects in increasingly sophisticated ways. Development aspects are easily removed from production builds. Production aspects are

intended to be used in both development and in production, but tend to affect only a few classes.

This section presents examples of aspects that can be used during development of Java applications. These aspects facilitate debugging, testing and performance tuning work. The aspects define behavior that ranges from simple tracing, to profiling, to testing of internal consistency within the application. Using AspectJ makes it possible to cleanly modularize this kind of functionality, thereby making it possible to easily enable and disable the functionality when desired.

4.2.2.1 Tracing

This first example shows how to increase the visibility of the internal workings of a program. It is a simple tracing aspect that prints a message at specified method calls. In our figure editor example, one such aspect might simply trace whenever points are drawn.

```
aspect SimpleTracing {
    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}
```

This code makes use of the `thisJoinPoint` special variable. Within all advice bodies this variable is bound to an object that describes the current join point. The effect of this code is to print a line like the following every time a figure element receives a `draw` method call:

```
Entering: call(void FigureElement.draw(GraphicsContext))
```

To understand the benefit of coding this with AspectJ consider changing the set of method calls that are traced. With AspectJ, this just requires editing the definition of the `tracedCalls` pointcut and recompiling. The individual methods that are traced do not need to be edited.

When debugging, programmers often invest considerable effort in figuring out a good set of trace points to use when looking for a particular kind of problem. When debugging is complete or appears to be complete it is frustrating to have to lose that investment by deleting trace statements from the code. The alternative of just commenting them out makes the code look bad, and can cause trace statements for one kind of debugging to get confused with trace statements for another kind of debugging.

With AspectJ it is easy to both preserve the work of designing a good set of trace points and disable the tracing when it isn't being used. This is done by writing an aspect specifically for that tracing mode, and removing that aspect from the compilation when it is not needed.

This ability to concisely implement and reuse debugging configurations that have proven useful in the past is a direct result of AspectJ modularizing a crosscutting design element

the set of methods that are appropriate to trace when looking for a given kind of information.

4.2.2.2 Profiling and Logging

Our second example shows you how to do some very specific profiling. Although many sophisticated profiling tools are available, and these can gather a variety of information and display the results in useful ways, you may sometimes want to profile or log some very specific behavior. In these cases, it is often possible to write a simple aspect similar to the ones above to do the job.

For example, the following aspect counts the number of calls to the `rotate` method on a `Line` and the number of calls to the `set*` methods of a `Point` that happen within the control flow of those calls to `rotate`:

```
aspect SetsInRotateCounting {
    int rotateCount = 0;
    int setCount = 0;

    before(): call(void Line.rotate(double)) {
        rotateCount++;
    }

    before(): call(void Point.set*(int))
        && cflow(call(void Line.rotate(double))) {
        setCount++;
    }
}
```

In effect, this aspect allows the programmer to ask very specific questions like

How many times is the `rotate` method defined on `Line` objects called?

and

How many times are methods defined on `Point` objects whose name begins with "set" called in fulfilling those rotate calls?

questions it may be difficult to express using standard profiling or logging tools.

4.2.2.3 Pre- and Post-Conditions

Many programmers use the "Design by Contract" style popularized by Bertand Meyer in *Object-Oriented Software Construction, 2/e*. In this style of programming, explicit pre-conditions test that callers of a method call it properly and explicit post-conditions test that methods properly do the work they are supposed to.

AspectJ makes it possible to implement pre- and post-condition testing in modular form. For example, this code

```
aspect PointBoundsChecking {

    pointcut setX(int x):
        (call(void FigureElement.setXY(int, int)) && args(x, *))
        || (call(void Point.setX(int)) && args(x));
```

```

pointcut setY(int y):
    (call(void FigureElement.setXY(int, int)) && args(*, y))
    || (call(void Point.setY(int)) && args(y));

before(int x): setX(x) {
    if ( x < MIN_X || x > MAX_X )
        throw new IllegalArgumentException("x is out of bounds.");
}

before(int y): setY(y) {
    if ( y < MIN_Y || y > MAX_Y )
        throw new IllegalArgumentException("y is out of bounds.");
}
}

```

implements the bounds checking aspect of pre-condition testing for operations that move points. Notice that the `setX` pointcut refers to all the operations that can set a `Point`'s `x` coordinate; this includes the `setX` method, as well as half of the `setXY` method. In this sense the `setX` pointcut can be seen as involving very fine-grained crosscutting — it names the `setX` method and half of the `setXY` method.

Even though pre- and post-condition testing aspects can often be used only during testing, in some cases developers may wish to include them in the production build as well. Again, because AspectJ makes it possible to modularize these crosscutting concerns cleanly, it gives developers good control over this decision.

4.2.2.4 Contract Enforcement

The property-based crosscutting mechanisms can be very useful in defining more sophisticated contract enforcement. One very powerful use of these mechanisms is to identify method calls that, in a correct program, should not exist. For example, the following aspect enforces the constraint that only the well-known factory methods can add an element to the registry of figure elements. Enforcing this constraint ensures that no figure element is added to the registry more than once.

```

static aspect RegistrationProtection {

    pointcut register(): call(void Registry.register(FigureElement));

    pointcut canRegister(): withincode(static *
FigureElement.make*(..));

    before(): register() && !canRegister() {
        throw new IllegalAccessException("Illegal call " +
thisJoinPoint);
    }
}

```

This aspect uses the `withincode` primitive pointcut to denote all join points that occur within the body of the factory methods on `FigureElement` (the methods with names that begin with "make"). This is a property-based pointcut because it identifies join points based not on their signature, but rather on the property that they occur specifically within the code of another method. The before advice declaration effectively says signal an error for any calls to `register` that are not within the factory methods.

This advice throws a runtime exception at certain join points, but AspectJ can do better. Using the `declare error` form, we can have the *compiler* signal the error.

```
static aspect RegistrationProtection {  
  
    pointcut register(): call(void Registry.register(FigureElement));  
    pointcut canRegister(): withincode(static *  
FigureElement.make*(..));  
  
    declare error: register() && !canRegister(): "Illegal call"  
}
```

When using this aspect, it is impossible for the compiler to compile programs with these illegal calls. This early detection is not always possible. In this case, since we depend only on static information (the `withincode` pointcut picks out join points totally based on their code, and the `call` pointcut here picks out join points statically). Other enforcement, such as the precondition enforcement, above, does require dynamic information such as the runtime value of parameters.

4.2.2.5 Configuration Management

Configuration management for aspects can be handled using a variety of make-file like techniques. To work with optional aspects, the programmer can simply define their make files to either include the aspect in the call to the AspectJ compiler or not, as desired.

Developers who want to be certain that no aspects are included in the production build can do so by configuring their make files so that they use a traditional Java compiler for production builds. To make it easy to write such make files, the AspectJ compiler has a command-line interface that is consistent with ordinary Java compilers.

4.2.3 Production Aspects

This section presents examples of aspects that are inherently intended to be included in the production builds of an application. Production aspects tend to add functionality to an application rather than merely adding more visibility of the internals of a program. Again, we begin with name-based aspects and follow with property-based aspects. Name-based production aspects tend to affect only a small number of methods. For this reason, they are a good next step for projects adopting AspectJ. But even though they tend to be small and simple, they can often have a significant effect in terms of making the program easier to understand and maintain.

4.2.3.1 Change Monitoring

The first example production aspect shows how one might implement some simple functionality where it is problematic to try and do it explicitly. It supports the code that refreshes the display. The role of the aspect is to maintain a dirty bit indicating whether or not an object has moved since the last time the display was refreshed.

Implementing this functionality as an aspect is straightforward. The `testAndClear` method is called by the display code to find out whether a figure element has moved recently. This method returns the current state of the dirty flag and resets it to false. The

`pointcut move` captures all the method calls that can move a figure element. The `after` advice on `move` sets the dirty flag whenever an object moves.

```
aspect MoveTracking {
    private static boolean dirty = false;

    public static boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;
    }

    pointcut move():
        call(void FigureElement.setXY(int, int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int));

    after() returning: move() {
        dirty = true;
    }
}
```

Even this simple example serves to illustrate some of the important benefits of using AspectJ in production code. Consider implementing this functionality with ordinary Java: there would likely be a helper class that contained the `dirty` flag, the `testAndClear` method, as well as a `setFlag` method. Each of the methods that could move a figure element would include a call to the `setFlag` method. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case.

The AspectJ implementation has several advantages over the standard implementation:

The structure of the crosscutting concern is captured explicitly. The `moves` pointcut clearly states all the methods involved, so the programmer reading the code sees not just individual calls to `setFlag`, but instead sees the real structure of the code. The IDE support included with AspectJ automatically reminds the programmer that this aspect advises each of the methods involved. The IDE support also provides commands to jump to the advice from the method and vice-versa.

Evolution is easier. If, for example, the aspect needs to be revised to record not just that some figure element moved, but rather to record exactly which figure elements moved, the change would be entirely local to the aspect. The pointcut would be updated to expose the object being moved, and the advice would be updated to record that object. The paper *An Overview of AspectJ* (available linked off of the AspectJ web site -- <http://eclipse.org/aspectj>), presented at ECOOP 2001, presents a detailed discussion of various ways this aspect could be expected to evolve.

The functionality is easy to plug in and out. Just as with development aspects, production aspects may need to be removed from the system, either because the functionality is no longer needed at all, or because it is not needed in certain configurations of a system. Because the functionality is modularized in a single aspect this is easy to do.

The implementation is more stable. If, for example, the programmer adds a subclass of `Line` that overrides the existing methods, this advice in this aspect will still apply. In the ordinary Java implementation the programmer would have to remember to add the call to `setFlag` in the new overriding method. This benefit is often even more compelling for property-based aspects (see the section [Providing Consistent Behavior](#)).

4.2.3.2 Context Passing

The crosscutting structure of context passing can be a significant source of complexity in Java programs. Consider implementing functionality that would allow a client of the figure editor (a program client rather than a human) to set the color of any figure elements that are created. Typically this requires passing a color, or a color factory, from the client, down through the calls that lead to the figure element factory. All programmers are familiar with the inconvenience of adding a first argument to a number of methods just to pass this kind of context information.

Using AspectJ, this kind of context passing can be implemented in a modular way. The following code adds after advice that runs only when the factory methods of `Figure` are called in the control flow of a method on a `ColorControllingClient`.

```
aspect ColorControl {
    pointcut CCClientCflow(ColorControllingClient client):
        cflow(call(* * (..)) && target(client));

    pointcut make(): call(FigureElement Figure.make*(..));

    after (ColorControllingClient c) returning (FigureElement fe):
        make() && CCClientCflow(c) {
        fe.setColor(c.colorFor(fe));
    }
}
```

This aspect affects only a small number of methods, but note that the non-AOP implementation of this functionality might require editing many more methods, specifically, all the methods in the control flow from the client to the factory. This is a benefit common to many property-based aspects while the aspect is short and affects only a modest number of benefits, the complexity the aspect saves is potentially much larger.

4.2.3.3 Providing Consistent Behavior

This example shows how a property-based aspect can be used to provide consistent handling of functionality across a large set of operations. This aspect ensures that all public methods of the `com.bigboxco` package log any Errors they throw to their caller (in Java, an Error is like an Exception, but it indicates that something really bad and usually unrecoverable has happened). The `publicMethodCall` pointcut captures the public method calls of the package, and the after advice runs whenever one of those calls throws an Error. The advice logs that Error and then the throw resumes.

```
aspect PublicErrorLogging {
    Log log = new Log();

    pointcut publicMethodCall():
```

```

        call(public * com.bigboxco.*.*(..));

        after() throwing (Error e): publicMethodCall() {
            log.write(e);
        }
    }
}

```

In some cases this aspect can log an exception twice. This happens if code inside the `com.bigboxco` package itself calls a public method of the package. In that case this code will log the error at both the outermost call into the `com.bigboxco` package and the re-entrant call. The `cflow` primitive pointcut can be used in a nice way to exclude these re-entrant calls:

```

after() throwing (Error e):
    publicMethodCall() && !cflow(publicMethodCall()) {
    log.write(e);
}

```

The following aspect is taken from work on the AspectJ compiler. The aspect advises about 35 methods in the `JavaParser` class. The individual methods handle each of the different kinds of elements that must be parsed. They have names like `parseMethodDec`, `parseThrows`, and `parseExpr`.

```

aspect ContextFilling {
    pointcut parse(JavaParser jp):
        call(* JavaParser.parse*(..))
        && target(jp)
        && !call(Stmt parseVarDec(boolean)); // var decs
                                           // are tricky

    around(JavaParser jp) returns ASTObject: parse(jp) {
        Token beginToken = jp.peekToken();
        ASTObject ret = proceed(jp);
        if (ret != null) jp.addContext(ret, beginToken);
        return ret;
    }
}

```

This example exhibits a property found in many aspects with large property-based pointcuts. In addition to a general property based pattern `call(* JavaParser.parse*(..))` it includes an exception to the pattern `!call(Stmt parseVarDec(boolean))`. The exclusion of `parseVarDec` happens because the parsing of variable declarations in Java is too complex to fit with the clean pattern of the other `parse*` methods. Even with the explicit exclusion this aspect is a clear expression of a clean crosscutting modularity. Namely that all `parse*` methods that return `ASTObjects`, except for `parseVarDec` share a common behavior for establishing the parse context of their result.

The process of writing an aspect with a large property-based pointcut, and of developing the appropriate exceptions can clarify the structure of the system. This is especially true, as in this case, when refactoring existing code to use aspects. When we first looked at the code for this aspect, we were able to use the IDE support provided in AJDE for JBuilder to see what methods the aspect was advising compared to our manual coding. We quickly

discovered that there were a dozen places where the aspect advice was in effect but we had not manually inserted the required functionality. Two of these were bugs in our prior non-AOP implementation of the parser. The other ten were needless performance optimizations. So, here, refactoring the code to express the crosscutting structure of the aspect explicitly made the code more concise and eliminated latent bugs.

4.2.4 Conclusion

AspectJ is a simple and practical aspect-oriented extension to Java. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns.

Adoption of AspectJ into an existing Java development project can be a straightforward and incremental task. One path is to begin by using only development aspects, going on to using production aspects and then reusable aspects after building up experience with AspectJ. Adoption can follow other paths as well. For example, some developers will benefit from using production aspects right away. Others may be able to write clean reusable aspects almost right away.

AspectJ enables both name-based and property based crosscutting. Aspects that use name-based crosscutting tend to affect a small number of other classes. But despite their small scale, they can often eliminate significant complexity compared to an ordinary Java implementation. Aspects that use property-based crosscutting can have small or large scale.

Using AspectJ results in clean well-modularized implementations of crosscutting concerns. When written as an AspectJ aspect the structure of a crosscutting concern is explicit and easy to understand. Aspects are also highly modular, making it possible to develop plug-and-play implementations of crosscutting functionality.

AspectJ provides more functionality than was covered by this short introduction. The next chapter, [The AspectJ Language](#), covers in detail more of the features of the AspectJ language. The following chapter, [Examples](#), then presents some carefully chosen examples that show you how AspectJ might be used. We recommend that you read the next two chapters carefully before deciding to adopt AspectJ into a project.

4.3 The AspectJ Language

The previous chapter, [Getting Started with AspectJ](#), was a brief overview of the AspectJ language. You should read this chapter to understand AspectJ's syntax and semantics. It covers the same material as the previous chapter, but more completely and in much more detail.

We will start out by looking at an example aspect that we'll build out of a pointcut, an introduction, and two pieces of advice. This example aspect will give us something concrete to talk about.

4.3.1 The Anatomy of an Aspect

This lesson explains the parts of AspectJ's aspects. By reading this lesson you will have an overview of what's in an aspect and you will be exposed to the new terminology introduced by AspectJ.

4.3.1.1 An Example Aspect

Here's an example of an aspect definition in AspectJ:

```
1 aspect FaultHandler {
2
3     private boolean Server.disabled = false;
4
5     private void reportFault() {
6         System.out.println("Failure! Please fix it.");
7     }
8
9     public static void fixServer(Server s) {
10         s.disabled = false;
11     }
12
13     pointcut services(Server s): target(s) && call(public * *(..));
14
15     before(Server s): services(s) {
16         if (s.disabled) throw new DisabledException();
17     }
18
19     after(Server s) throwing (FaultException e): services(s) {
20         s.disabled = true;
21         reportFault();
22     }
23 }
```

The `FaultHandler` consists of one inter-type field on `Server` (line 03), two methods (lines 05-07 and 09-11), one pointcut definition (line 13), and two pieces of advice (lines 15-17 and 19-22).

This covers the basics of what aspects can contain. In general, aspects consist of an association of other program entities, ordinary variables and methods, pointcut definitions, inter-type declarations, and advice, where advice may be before, after or around advice. The remainder of this lesson focuses on those crosscut-related constructs.

4.3.1.2 Pointcuts

AspectJ's pointcut definitions give names to pointcuts. Pointcuts themselves pick out join points, i.e. interesting points in the execution of a program. These join points can be method or constructor invocations and executions, the handling of exceptions, field assignments and accesses, etc. Take, for example, the pointcut definition in line 13:

```
pointcut services(Server s): target(s) && call(public * *(..))
```

This pointcut, named `services`, picks out those points in the execution of the program when `Server` objects have their public methods called. It also allows anyone using the `services` pointcut to access the `Server` object whose method is being called.

The idea behind this pointcut in the `ExceptionHandler` aspect is that fault-handling-related behavior must be triggered on the calls to public methods. For example, the server may be unable to proceed with the request because of some fault. The calls of those methods are, therefore, interesting events for this aspect, in the sense that certain fault-related things will happen when these events occur.

Part of the context in which the events occur is exposed by the formal parameters of the pointcut. In this case, that consists of objects of type `Server`. That formal parameter is then being used on the right hand side of the declaration in order to identify which events the pointcut refers to. In this case, a pointcut picking out join points where a `Server` is the target of some operation (`target(s)`) is being composed (`&&`, meaning and) with a pointcut picking out call join points (`call(...)`). The calls are identified by signatures that can include wild cards. In this case, there are wild cards in the return type position (first `*`), in the name position (second `*`) and in the argument list position (`..`); the only concrete information is given by the qualifier `public`.

Pointcuts pick out arbitrarily large numbers of join points of a program. But they pick out only a small number of *kinds* of join points. Those kinds of join points correspond to some of the most important concepts in Java. Here is an incomplete list: method call, method execution, exception handling, instantiation, constructor execution, and field access. Each kind of join point can be picked out by its own specialized pointcut that you will learn about in other parts of this guide.

4.3.1.3 Advice

A piece of advice brings together a pointcut and a body of code to define aspect implementation that runs at join points picked out by the pointcut. For example, the advice in lines 15-17 specifies that the following piece of code

```
{
    if (s.disabled) throw new DisabledException();
}
```

is executed when instances of the `Server` class have their public methods called, as specified by the pointcut `services`. More specifically, it runs when those calls are made, just before the corresponding methods are executed.

The advice in lines 19-22 defines another piece of implementation that is executed on the same pointcut:

```
{
    s.disabled = true;
    reportFault();
}
```

But this second method executes after those operations throw exception of type `FaultException`.

There are two other variations of after advice: upon successful return and upon return, either successful or with an exception. There is also a third kind of advice called around. You will see those in other parts of this guide.

4.3.2 Join Points and Pointcuts

Consider the following Java class:

```
class Point {
    private int x, y;

    Point(int x, int y) { this.x = x; this.y = y; }

    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }

    int getX() { return x; }
    int getY() { return y; }
}
```

In order to get an intuitive understanding of AspectJ's join points and pointcuts, let's go back to some of the basic principles of Java. Consider the following a method declaration in class `Point`:

```
void setX(int x) { this.x = x; }
```

This piece of program says that that when method named `setX` with an `int` argument called on an object of type `Point`, then the method body `{ this.x = x; }` is executed. Similarly, the constructor of the class states that when an object of type `Point` is instantiated through a constructor with two `int` arguments, then the constructor body `{ this.x = x; this.y = y; }` is executed.

One pattern that emerges from these descriptions is

When something happens, then something gets executed.

In object-oriented programs, there are several kinds of "things that happen" that are determined by the language. We call these the join points of Java. Join points consist of things like method calls, method executions, object instantiations, constructor executions, field references and handler executions. (See the [AspectJ Quick Reference](#) for a complete listing.)

Pointcuts pick out these join points. For example, the pointcut

```
pointcut setter(): target(Point) &&
    (call(void setX(int)) ||
     call(void setY(int)));
```

picks out each call to `setX(int)` or `setY(int)` when called on an instance of `Point`. Here's another example:

```
pointcut ioHandler(): within(MyClass) && handler(IOException);
```

This pointcut picks out each the join point when exceptions of type `IOException` are handled inside the code defined by class `MyClass`.

Pointcut definitions consist of a left-hand side and a right-hand side, separated by a colon. The left-hand side consists of the pointcut name and the pointcut parameters (i.e. the data available when the events happen). The right-hand side consists of the pointcut itself.

4.3.2.1 Some Example Pointcuts

Here are examples of pointcuts picking out

when a particular method body executes

```
execution(void Point.setX(int))
```

when a method is called

```
call(void Point.setX(int))
```

when an exception handler executes

```
handler(ArrayOutOfBoundsException)
```

when the object currently executing (i.e. `this`) is of type `SomeType`

```
this(SomeType)
```

when the target object is of type `SomeType`

```
target(SomeType)
```

when the executing code belongs to class `MyClass`

```
within(MyClass)
```

when the join point is in the control flow of a call to a `Test`'s no-argument `main` method

```
cflow(call(void Test.main()))
```

Pointcuts compose through the operations `or ("||")`, `and ("&&")` and `not ("!")`.

It is possible to use wildcards. So

```
execution(* *(..))
```

```
call(* set(..))
```

means (1) the execution of any method regardless of return or parameter types, and (2) the call to any method named `set` regardless of return or parameter types - in case of overloading there may be more than one such `set` method; this pointcut picks out calls to all of them.

You can select elements based on types. For example,

```
execution(int *())
```

```
call(* setY(long))
```

```
call(* Point.setY(int))
```

```
call(*.new(int, int))
```

means (1) the execution of any method with no parameters that returns an `int`, (2) the call to any `setY` method that takes a `long` as an argument, regardless of return type or declaring type, (3) the call to any of `Point`'s `setY` methods that take an `int` as an argument, regardless of return type, and (4) the call to any classes' constructor, so long as it takes exactly two `ints` as arguments.

You can compose pointcuts. For example,

```
target(Point) && call(int *())
call(* *(..)) && (within(Line) || within(Point))
within(*) && execution(*.new(int))
!this(Point) && call(int *(..))
```

means (1) any call to an `int` method with no arguments on an instance of `Point`, regardless of its name, (2) any call to any method where the call is made from the code in `Point`'s or `Line`'s type declaration, (3) the execution of any constructor taking exactly one `int` argument, regardless of where the call is made from, and (4) any method call to an `int` method when the executing object is any type except `Point`.

You can select methods and constructors based on their modifiers and on negations of modifiers. For example, you can say:

```
call(public * *(..))
execution(!static * *(..))
execution(public !static * *(..))
```

which means (1) any call to a public method, (2) any execution of a non-static method, and (3) any execution of a public, non-static method.

Pointcuts can also deal with interfaces. For example, given the interface

- - `interface MyInterface { ... }`

the pointcut `call(* MyInterface.*(..))` picks out any call to a method in `MyInterface`'s signature -- that is, any method defined by `MyInterface` or inherited by one of its supertypes.

4.3.2.2 call vs. execution

When methods and constructors run, there are two interesting times associated with them. That is when they are called, and when they actually execute.

AspectJ exposes these times as call and execution join points, respectively, and allows them to be picked out specifically by `call` and `execution` pointcuts.

So what's the difference between these join points? Well, there are a number of differences:

Firstly, the lexical pointcut declarations `within` and `withincode` match differently. At a call join point, the enclosing code is that of the call site. This means that `call(void m()) && withincode(void m())` will only capture directly recursive calls, for example.

At an execution join point, however, the program is already executing the method, so the enclosing code is the method itself: `execution(void m())` && `withincode(void m())` is the same as `execution(void m())`.

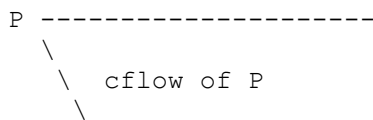
Secondly, the call join point does not capture super calls to non-static methods. This is because such super calls are different in Java, since they don't behave via dynamic dispatch like other calls to non-static methods.

The rule of thumb is that if you want to pick a join point that runs when an actual piece of code runs (as is often the case for tracing), use `execution`, but if you want to pick one that runs when a particular *signature* is called (as is often the case for production aspects), use `call`.

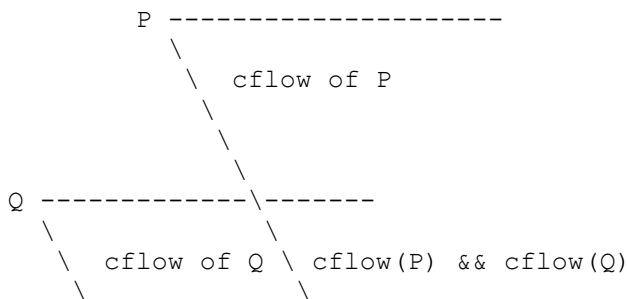
4.3.2.3 Pointcut composition

Pointcuts are put together with the operators `&` (spelled `&&`), `|` (spelled `||`), and `!` (spelled `!`). This allows the creation of very powerful pointcuts from the simple building blocks of primitive pointcuts. This composition can be somewhat confusing when used with primitive pointcuts like `cflow` and `cflowbelow`. Here's an example:

`cflow(P)` picks out each join point in the control flow of the join points picked out by P . So, pictorially:

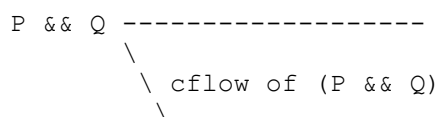


What does $\text{cflow}(P) \ \&\& \ \text{cflow}(Q)$ pick out? Well, it picks out each join point that is in both the control flow of P and in the control flow of Q . So...



Note that P and Q might not have any join points in common... but their control flows might have join points in common.

But what does `cflow(P && Q)` mean? Well, it means the control flow of those join points that are both picked out by P and picked out by Q .



and if there are *no* join points that are both picked by P and picked out by Q , then there's no chance that there are any join points in the control flow of $(P \ \&\& \ Q)$.

Here's some code that expresses this.

```
public class Test {
    public static void main(String[] args) {
        foo();
    }
    static void foo() {
        goo();
    }
    static void goo() {
        System.out.println("hi");
    }
}

aspect A {
    pointcut fooPC(): execution(void Test.foo());
    pointcut gooPC(): execution(void Test.goo());
    pointcut printPC(): call(void java.io.PrintStream.println(String));

    before(): cflow(fooPC()) && cflow(gooPC()) && printPC() {
        System.out.println("should occur");
    }

    before(): cflow(fooPC() && gooPC()) && printPC() {
        System.out.println("should not occur");
    }
}
```

4.3.2.4 Pointcut Parameters

Consider again the first pointcut definition in this chapter:

```
pointcut setter(): target(Point) &&
    (call(void setX(int)) ||
     call(void setY(int)));
```

As we've seen, this pointcut picks out each call to `setX(int)` or `setY(int)` methods where the target is an instance of `Point`. The pointcut is given the name `setters` and no parameters on the left-hand side. An empty parameter list means that none of the context from the join points is published from this pointcut. But consider another version of version of this pointcut definition:

```
pointcut setter(Point p): target(p) &&
    (call(void setX(int)) ||
     call(void setY(int)));
```

This version picks out exactly the same join points. But in this version, the pointcut has one parameter of type `Point`. This means that any advice that uses this pointcut has access to a `Point` from each join point picked out by the pointcut. Inside the pointcut definition this `Point` is named `p` is available, and according to the right-hand side of the definition, that `Point p` comes from the `target` of each matched join point.

Here's another example that illustrates the flexible mechanism for defining pointcut parameters:

```
pointcut testEquality(Point p): target(Point) &&
    args(p) &&
    call(boolean equals(Object));
```

This pointcut also has a parameter of type `Point`. Similar to the `setters` pointcut, this means that anyone using this pointcut has access to a `Point` from each join point. But in this case, looking at the right-hand side we find that the object named in the parameters is not the target `Point` object that receives the call; it's the argument (also of type `Point`) passed to the `equals` method when some other `Point` is the target. If we wanted access to both `Points`, then the pointcut definition that would expose target `Point` `p1` and argument `Point` `p2` would be

```
pointcut testEquality(Point p1, Point p2): target(p1) &&
    args(p2) &&
    call(boolean
equals(Object));
```

Let's look at another variation of the `setters` pointcut:

```
pointcut setter(Point p, int newval): target(p) &&
    args(newval) &&
    (call(void setX(int)) ||
    call(void setY(int)));
```

In this case, a `Point` object and an `int` value are exposed by the named pointcut. Looking at the the right-hand side of the definition, we find that the `Point` object is the target object, and the `int` value is the called method's argument.

The use of pointcut parameters is relatively flexible. The most important rule is that all the pointcut parameters must be bound at every join point picked out by the pointcut. So, for example, the following pointcut definition will result in a compilation error:

```
pointcut badPointcut(Point p1, Point p2):
    (target(p1) && call(void setX(int))) ||
    (target(p2) && call(void setY(int)));
```

because `p1` is only bound when calling `setX`, and `p2` is only bound when calling `setY`, but the pointcut picks out all of these join points and tries to bind both `p1` and `p2`.

4.3.2.5 Example: `HandleLiveness`

The example below consists of two object classes (plus an exception class) and one aspect. `Handle` objects delegate their public, non-static operations to their `Partner` objects. The aspect `HandleLiveness` ensures that, before the delegations, the partner exists and is alive, or else it throws an exception.

```
class Handle {
    Partner partner = new Partner();

    public void foo() { partner.foo(); }
```

```

    public void bar(int x) { partner.bar(x); }

    public static void main(String[] args) {
        Handle h1 = new Handle();
        h1.foo();
        h1.bar(2);
    }
}

class Partner {
    boolean isAlive() { return true; }
    void foo() { System.out.println("foo"); }
    void bar(int x) { System.out.println("bar " + x); }
}

aspect HandleLiveness {
    before(Handle handle): target(handle) && call(public * *(..)) {
        if ( handle.partner == null || !handle.partner.isAlive() ) {
            throw new DeadPartnerException();
        }
    }
}

class DeadPartnerException extends RuntimeException {}

```

4.3.3 Advice

Advice defines pieces of aspect implementation that execute at well-defined points in the execution of the program. Those points can be given either by named pointcuts (like the ones you've seen above) or by anonymous pointcuts. Here is an example of an advice on a named pointcut:

```

pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                   (call(void setX(int) ||
                                   call(void setY(int)));

before(Point p1, int newval): setter(p1, newval) {
    System.out.println("About to set something in " + p1 +
        " to the new value " + newval);
}

```

And here is exactly the same example, but using an anonymous pointcut:

```

before(Point p1, int newval): target(p1) && args(newval)
                                   (call(void setX(int)) ||
                                   call(void setY(int))) {
    System.out.println("About to set something in " + p1 +
        " to the new value " + newval);
}

```

Here are examples of the different advice:

This before advice runs just before the join points picked out by the (anonymous) pointcut:

```

before(Point p, int x): target(p) && args(x) && call(void setX(int))
{
    if (!p.assertX(x)) return;
}

```

This after advice runs just after each join point picked out by the (anonymous) pointcut, regardless of whether it returns normally or throws an exception:

```

after(Point p, int x): target(p) && args(x) && call(void setX(int)) {
    if (!p.assertX(x)) throw new PostConditionViolation();
}

```

This after returning advice runs just after each join point picked out by the (anonymous) pointcut, but only if it returns normally. The return value can be accessed, and is named `x` here. After the advice runs, the return value is returned:

```

after(Point p) returning(int x): target(p) && call(int getX()) {
    System.out.println("Returning int value " + x + " for p = " + p);
}

```

This after throwing advice runs just after each join point picked out by the (anonymous) pointcut, but only when it throws an exception of type `Exception`. Here the exception value can be accessed with the name `e`. The advice re-raises the exception after it's done:

```

after() throwing(Exception e): target(Point) && call(void setX(int))
{
    System.out.println(e);
}

```

This around advice traps the execution of the join point; it runs *instead* of the join point. The original action associated with the join point can be invoked through the special `proceed` call:

```

void around(Point p, int x): target(p)
    && args(x)
    && call(void setX(int)) {
    if (p.assertX(x)) proceed(p, x);
    p.releaseResources();
}

```

4.3.4 Inter-type declarations

Aspects can declare members (fields, methods, and constructors) that are owned by other types. These are called inter-type members. Aspects can also declare that other types implement new interfaces or extend a new class. Here are examples of some such inter-type declarations:

This declares that each `Server` has a boolean field named `disabled`, initialized to `false`:

```

private boolean Server.disabled = false;

```

It is declared `private`, which means that it is private *to the aspect*: only code in the aspect can see the field. And even if `Server` has another private field named `disabled`

(declared in `Server` or in another aspect) there won't be a name collision, since no reference to `disabled` will be ambiguous.

This declares that each `Point` has an `int` method named `getX` with no arguments that returns whatever `this.x` is:

```
public int Point.getX() { return this.x; }
```

Inside the body, `this` is the `Point` object currently executing. Because the method is publically declared any code can call it, but if there is some other `Point.getX()` declared there will be a compile-time conflict.

This publically declares a two-argument constructor for `Point`:

```
public Point.new(int x, int y) { this.x = x; this.y = y; }
```

This publicly declares that each `Point` has an `int` field named `x`, initialized to zero:

```
public int Point.x = 0;
```

Because this is publically declared, it is an error if `Point` already has a field named `x` (defined by `Point` or by another aspect).

This declares that the `Point` class implements the `Comparable` interface:

```
declare parents: Point implements Comparable;
```

Of course, this will be an error unless `Point` defines the methods required by `Comparable`.

This declares that the `Point` class extends the `GeometricObject` class.

```
declare parents: Point extends GeometricObject;
```

An aspect can have several inter-type declarations. For example, the following declarations

```
public String Point.name;
public void Point.setName(String name) { this.name = name; }
```

publicly declare that `Point` has both a `String` field `name` and a `void` method `setName(String)` (which refers to the `name` field declared by the aspect).

An inter-type member can only have one target type, but often you may wish to declare the same member on more than one type. This can be done by using an inter-type member in combination with a private interface:

```
aspect A {
  private interface HasName {}
  declare parents: (Point || Line || Square) implements HasName;

  private String HasName.name;
  public String HasName.getName() { return name; }
}
```

This declares a marker interface `HasName`, and also declares that any type that is either `Point`, `Line`, or `Square` implements that interface. It also privately declares that all `HasName` object have a `String` field called `name`, and publically declares that all `HasName`

objects have a `String` method `getName()` (which refers to the privately declared `name` field).

As you can see from the above example, an aspect can declare that interfaces have fields and methods, even non-constant fields and methods with bodies.

4.3.4.1 Inter-type Scope

AspectJ allows private and package-protected (default) inter-type declarations in addition to public inter-type declarations. Private means private in relation to the aspect, not necessarily the target type. So, if an aspect makes a private inter-type declaration of a field

```
private int Foo.x;
```

Then code in the aspect can refer to `Foo`'s `x` field, but nobody else can. Similarly, if an aspect makes a package-protected introduction,

```
int Foo.x;
```

then everything in the aspect's package (which may or may not be `Foo`'s package) can access `x`.

4.3.4.2 Example: `PointAssertions`

The example below consists of one class and one aspect. The aspect privately declares the assertion methods of `Point`, `assertX` and `assertY`. It also guards calls to `setX` and `setY` with calls to these assertion methods. The assertion methods are declared privately because other parts of the program (including the code in `Point`) have no business accessing the `assert` methods. Only the code inside of the aspect can call those methods.

```
class Point {
    int x, y;

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    public static void main(String[] args) {
        Point p = new Point();
        p.setX(3); p.setY(333);
    }
}

aspect PointAssertions {

    private boolean Point.assertX(int x) {
        return (x <= 100 && x >= 0);
    }
    private boolean Point.assertY(int y) {
        return (y <= 100 && y >= 0);
    }

    before(Point p, int x): target(p) && args(x) && call(void
setX(int)) {
        if (!p.assertX(x)) {
```

```

        System.out.println("Illegal value for x"); return;
    }
}
before(Point p, int y): target(p) && args(y) && call(void
setY(int)) {
    if (!p.assertY(y)) {
        System.out.println("Illegal value for y"); return;
    }
}
}

```

4.3.5 thisJoinPoint

AspectJ provides a special reference variable, `thisJoinPoint`, that contains reflective information about the current join point for the advice to use. The `thisJoinPoint` variable can only be used in the context of advice, just like `this` can only be used in the context of non-static methods and variable initializers. In advice, `thisJoinPoint` is an object of type org.aspectj.lang.JoinPoint.

One way to use it is simply to print it out. Like all Java objects, `thisJoinPoint` has a `toString()` method that makes quick-and-dirty tracing easy:

```

class TraceNonStaticMethods {
    before(Point p): target(p) && call(* *(..)) {
        System.out.println("Entering " + thisJoinPoint + " in " + p);
    }
}

```

The type of `thisJoinPoint` includes a rich reflective class hierarchy of signatures, and can be used to access both static and dynamic information about join points such as the arguments of the join point:

```
thisJoinPoint.getArgs()
```

In addition, it holds an object consisting of all the static information about the join point such as corresponding line number and static signature:

```
thisJoinPoint.getStaticPart()
```

If you only need the static information about the join point, you may access the static part of the join point directly with the special variable `thisJoinPointStaticPart`. Using `thisJoinPointStaticPart` will avoid the run-time creation of the join point object that may be necessary when using `thisJoinPoint` directly.

It is always the case that

```

thisJoinPointStaticPart == thisJoinPoint.getStaticPart()

thisJoinPoint.getKind() == thisJoinPointStaticPart.getKind()
thisJoinPoint.getSignature() ==
thisJoinPointStaticPart.getSignature()
thisJoinPoint.getSourceLocation() ==
thisJoinPointStaticPart.getSourceLocation()

```


One more reflective variable is available: `thisEnclosingJoinPointStaticPart`. This, like `thisJoinPointStaticPart`, only holds the static part of a join point, but it is not the current but the enclosing join point. So, for example, it is possible to print out the calling source location (if available) with

```
before() : execution (* *(..)) {  
  
System.err.println(thisEnclosingJoinPointStaticPart.getSourceLocation()  
)  
}
```

4.4 Examples

This chapter consists entirely of examples of AspectJ use.

The examples can be grouped into four categories:

- technique** Examples which illustrate how to use one or more features of the language.
- development** Examples of using AspectJ during the development phase of a project.
- production** Examples of using AspectJ to provide functionality in an application.
- reusable** Examples of reuse of aspects and pointcuts.

4.4.1 Obtaining, Compiling and Running the Examples

The examples source code is part of the AspectJ distribution which may be downloaded from the AspectJ project page (<http://eclipse.org/aspectj>).

Compiling most examples is straightforward. Go the *InstallDir/examples* directory, and look for a *.lst* file in one of the example subdirectories. Use the *-arglist* option to *ajc* to compile the example. For instance, to compile the telecom example with billing, type

```
ajc -argfile telecom/billing.lst
```

To run the examples, your classpath must include the AspectJ run-time Java archive (*aspectjrt.jar*). You may either set the *CLASSPATH* environment variable or use the *-classpath* command line option to the Java interpreter:

```
(In Unix use a : in the CLASSPATH)  
java -classpath ".:InstallDir/lib/aspectjrt.jar"  
telecom.billingSimulation
```

```
(In Windows use a ; in the CLASSPATH)  
java -classpath ".;InstallDir/lib/aspectjrt.jar"  
telecom.billingSimulation
```

4.4.2 Basic Techniques

This section presents two basic techniques of using AspectJ, one each from the two fundamental ways of capturing crosscutting concerns: with dynamic join points and advice, and with static introduction. Advice changes an application's behavior. Introduction changes both an application's behavior and its structure.

The first example, [the section called “Join Points and `thisJoinPoint`”](#), is about gathering and using information about the join point that has triggered some advice. The second example, [the section called “Roles and Views”](#), concerns a crosscutting view of an existing class hierarchy.

4.4.2.1 Join Points and `thisJoinPoint`

(The code for this example is in *InstallDir/examples/tjp*.)

A join point is some point in the execution of a program together with a view into the execution context when that point occurs. Join points are picked out by pointcuts. When a program reaches a join point, advice on that join point may run in addition to (or instead of) the join point itself.

When using a pointcut that picks out join points of a single kind by name, typically the advice will know exactly what kind of join point it is associated with. The pointcut may even publish context about the join point. Here, for example, since the only join points picked out by the pointcut are calls of a certain method, we can get the target value and one of the argument values of the method calls directly.

```
before(Point p, int x): target(p)
                        && args(x)
                        && call(void setX(int)) {
    if (!p.assertX(x)) {
        System.out.println("Illegal value for x"); return;
    }
}
```

But sometimes the shape of the join point is not so clear. For instance, suppose a complex application is being debugged, and we want to trace when any method of some class is executed. The pointcut

```
pointcut execsInProblemClass(): within(ProblemClass)
                                && execution(* *(..));
```

will pick out each execution join point of every method defined within `ProblemClass`. Since advice executes at each join point picked out by the pointcut, we can reasonably ask which join point was reached.

Information about the join point that was matched is available to advice through the special variable `thisJoinPoint`, of type [org.aspectj.lang.JoinPoint](#). Through this object we can access information such as

- the kind of join point that was matched
- the source location of the code associated with the join point
- normal, short and long string representations of the current join point
- the actual argument values of the join point

- the signature of the member associated with the join point
- the currently executing object
- the target object
- an object encapsulating the static information about the join point. This is also available through the special variable `thisJoinPointStaticPart`.

4.4.2.1.1 *The Demo class*

The class `tjp.Demo` in `tjp/Demo.java` defines two methods `foo` and `bar` with different parameter lists and return types. Both are called, with suitable arguments, by `Demo`'s `go` method which was invoked from within its `main` method.

```
public class Demo {
    static Demo d;

    public static void main(String[] args){
        new Demo().go();
    }

    void go(){
        d = new Demo();
        d.foo(1,d);
        System.out.println(d.bar(new Integer(3)));
    }

    void foo(int i, Object o){
        System.out.println("Demo.foo(" + i + ", " + o + ")\n");
    }

    String bar (Integer j){
        System.out.println("Demo.bar(" + j + ")\n");
        return "Demo.bar(" + j + ")";
    }
}
```

4.4.2.1.2 *The GetInfo aspect*

This aspect uses around advice to intercept the execution of methods `foo` and `bar` in `Demo`, and prints out information garnered from `thisJoinPoint` to the console.

```
aspect GetInfo {

    static final void println(String s){ System.out.println(s); }

    pointcut goCut(): cflow(this(Demo) && execution(void go()));

    pointcut demoExecs(): within(Demo) && execution(* *(..));

    Object around(): demoExecs() && !execution(* go()) && goCut() {
        println("Intercepted message: " +
            thisJoinPointStaticPart.getSignature().getName());
        println("in class: " +
            thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
    }
}
```

```

        printParameters(thisJoinPoint);
        println("Running original method: \n" );
        Object result = proceed();
        println("  result: " + result );
        return result;
    }

    static private void printParameters(JoinPoint jp) {
        println("Arguments: " );
        Object[] args = jp.getArgs();
        String[] names =
        ((CodeSignature)jp.getSignature()).getParameterNames();
        Class[] types =
        ((CodeSignature)jp.getSignature()).getParameterTypes();
        for (int i = 0; i < args.length; i++) {
            println("  " + i + ". " + names[i] +
                " : " + types[i].getName() +
                " = " + args[i]);
        }
    }
}

```

4.4.2.1.2.1 Defining the scope of a pointcut

The pointcut `goCut` is defined as

```
cflow(this(Demo)) && execution(void go())
```

so that only executions made in the control flow of `Demo.go` are intercepted. The control flow from the method `go` includes the execution of `go` itself, so the definition of the around advice includes `!execution(* go())` to exclude it from the set of executions advised.

4.4.2.1.2.2 Printing the class and method name

The name of the method and that method's defining class are available as parts of the org.aspectj.lang.Signature object returned by calling `getSignature()` on either `thisJoinPoint` OR `thisJoinPointStaticPart`.

4.4.2.1.2.3 Printing the parameters

The static portions of the parameter details, the name and types of the parameters, can be accessed through the org.aspectj.lang.reflect.CodeSignature associated with the join point. All execution join points have code signatures, so the cast to `CodeSignature` cannot fail.

The dynamic portions of the parameter details, the actual values of the parameters, are accessed directly from the execution join point object.

4.4.2.2 Roles and Views

(The code for this example is in `InstallDir/examples/introduction`.)

Like advice, inter-type declarations are members of an aspect. They declare members that act as if they were defined on another class. Unlike advice, inter-type declarations affect

not only the behavior of the application, but also the structural relationship between an application's classes.

This is crucial: Publically affecting the class structure of an application makes these modifications available to other components of the application.

Aspects can declare inter-type

- fields
- methods
- constructors

and can also declare that target types

- implement new interfaces
- extend new classes

This example provides three illustrations of the use of inter-type declarations to encapsulate roles or views of a class. The class our aspect will be dealing with, `Point`, is a simple class with rectangular and polar coordinates. Our inter-type declarations will make the class `Point`, in turn, cloneable, hashable, and comparable. These facilities are provided by AspectJ without having to modify the code for the class `Point`.

4.4.2.2.1 The `Point` class

The `Point` class defines geometric points whose interface includes polar and rectangular coordinates, plus some simple operations to relocate points. `Point`'s implementation has attributes for both its polar and rectangular coordinates, plus flags to indicate which currently reflect the position of the point. Some operations cause the polar coordinates to be updated from the rectangular, and some have the opposite effect. This implementation, which is intended to give the minimum number of conversions between coordinate systems, has the property that not all the attributes stored in a `Point` object are necessary to give a canonical representation such as might be used for storing, comparing, cloning or making hash codes from points. Thus the aspects, though simple, are not totally trivial.

The diagram below gives an overview of the aspects and their interaction with the class `Point`.

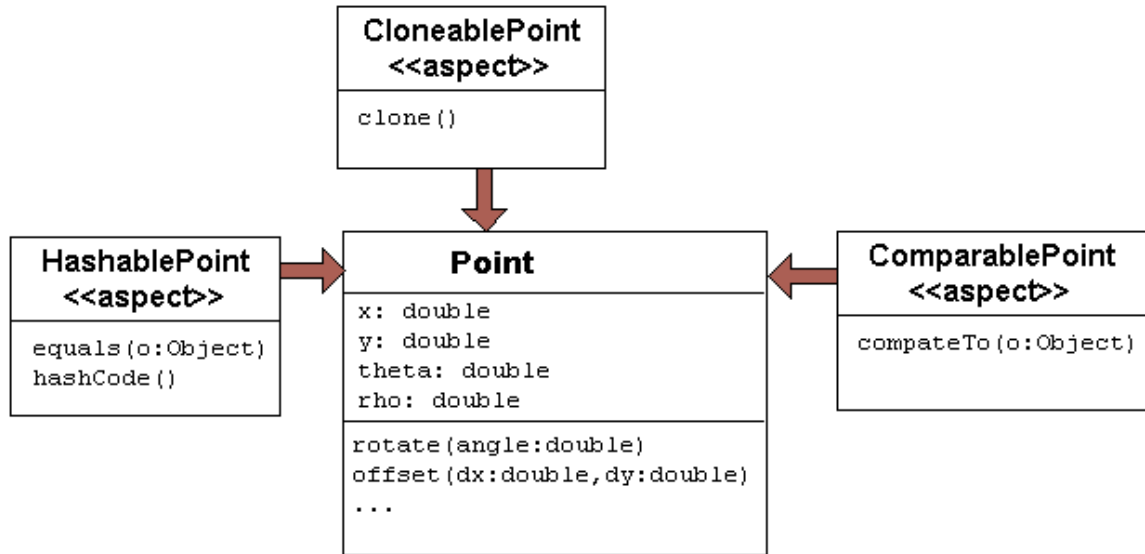


Figure 11 Roles for Point

4.4.2.2.2 The *CloneablePoint* aspect

This first aspect is responsible for `Point`'s implementation of the `Cloneable` interface. It declares that `Point` implements `Cloneable` with a `declare parents` form, and also publically declares a specialized `Point`'s `clone()` method. In Java, all objects inherit the method `clone` from the class `Object`, but an object is not cloneable unless its class also implements the interface `Cloneable`. In addition, classes frequently have requirements over and above the simple bit-for-bit copying that `Object.clone` does. In our case, we want to update a `Point`'s coordinate systems before we actually clone the `Point`. So our aspect makes sure that `Point` overrides `Object.clone` with a new method that does what we want.

We also define a test `main` method in the aspect for convenience.

```

public aspect CloneablePoint {

    declare parents: Point implements Cloneable;

    public Object Point.clone() throws CloneNotSupportedException {
        // we choose to bring all fields up to date before cloning.
        makeRectangular();
        makePolar();
        return super.clone();
    }

    public static void main(String[] args) {
        Point p1 = new Point();
        Point p2 = null;

        p1.setPolar(Math.PI, 1.0);
        try {
            p2 = (Point)p1.clone();
        } catch (CloneNotSupportedException e) {}
        System.out.println("p1 =" + p1 );
    }
}
  
```

```

        System.out.println("p2 =" + p2 );

        p1.rotate(Math.PI / -2);
        System.out.println("p1 =" + p1 );
        System.out.println("p2 =" + p2 );
    }
}

```

4.4.2.2.3 *The ComparablePoint aspect*

`ComparablePoint` is responsible for `Point`'s implementation of the `Comparable` interface.

The interface `Comparable` defines the single method `compareTo` which can be use to define a natural ordering relation among the objects of a class that implement it.

`ComparablePoint` uses declare parents to declare that `Point` implements `Comparable`, and also publically declares the appropriate `compareTo(Object)` method: A `Point` `p1` is said to be less than another `Point` `p2` if `p1` is closer to the origin.

We also define a test `main` method in the aspect for convenience.

```

public aspect ComparablePoint {

    declare parents: Point implements Comparable;

    public int Point.compareTo(Object o) {
        return (int) (this.getRho() - ((Point)o).getRho());
    }

    public static void main(String[] args){
        Point p1 = new Point();
        Point p2 = new Point();

        System.out.println("p1 == p2 :" + p1.compareTo(p2));

        p1.setRectangular(2,5);
        p2.setRectangular(2,5);
        System.out.println("p1 == p2 :" + p1.compareTo(p2));

        p2.setRectangular(3,6);
        System.out.println("p1 == p2 :" + p1.compareTo(p2));

        p1.setPolar(Math.PI, 4);
        p2.setPolar(Math.PI, 4);
        System.out.println("p1 == p2 :" + p1.compareTo(p2));

        p1.rotate(Math.PI / 4.0);
        System.out.println("p1 == p2 :" + p1.compareTo(p2));

        p1.offset(1,1);
        System.out.println("p1 == p2 :" + p1.compareTo(p2));
    }
}

```

4.4.2.2.4 *The HashablePoint aspect*

Our third aspect is responsible for `Point`'s overriding of `Object`'s `equals` and `hashCode` methods in order to make `Points` hashable.

The method `Object.hashCode` returns a unique integer, suitable for use as a hash table key. Different implementations are allowed to return different integers, but must return distinct integers for distinct objects, and the same integer for objects that test equal. But since the default implementation of `Object.equals` returns `true` only when two objects are identical, we need to redefine both `equals` and `hashCode` to work correctly with objects of type `Point`. For example, we want two `Point` objects to test equal when they have the same `x` and `y` values, or the same `rho` and `theta` values, not just when they refer to the same object. We do this by overriding the methods `equals` and `hashCode` in the class `Point`.

So `HashablePoint` declares `Point`'s `hashCode` and `equals` methods, using `Point`'s rectangular coordinates to generate a hash code and to test for equality. The `x` and `y` coordinates are obtained using the appropriate `get` methods, which ensure the rectangular coordinates are up-to-date before returning their values.

And again, we supply a `main` method in the aspect for testing.

```
public aspect HashablePoint {

    public int Point.hashCode() {
        return (int) (getX() + getY() % Integer.MAX_VALUE);
    }

    public boolean Point.equals(Object o) {
        if (o == this) { return true; }
        if (!(o instanceof Point)) { return false; }
        Point other = (Point)o;
        return (getX() == other.getX()) && (getY() == other.getY());
    }

    public static void main(String[] args) {
        Hashtable h = new Hashtable();
        Point p1 = new Point();

        p1.setRectangular(10, 10);
        Point p2 = new Point();

        p2.setRectangular(10, 10);

        System.out.println("p1 = " + p1);
        System.out.println("p2 = " + p2);
        System.out.println("p1.hashCode() = " + p1.hashCode());
        System.out.println("p2.hashCode() = " + p2.hashCode());

        h.put(p1, "P1");
        System.out.println("Got: " + h.get(p2));
    }
}
```

4.4.3 Development Aspects

The following aspects are useful during development.

4.4.3.1 Tracing using aspects

(The code for this example is in *InstallDir/examples/tracing*.)

Writing a class that provides tracing functionality is easy: a couple of functions, a boolean flag for turning tracing on and off, a choice for an output stream, maybe some code for formatting the output -- these are all elements that `Trace` classes have been known to have. `Trace` classes may be highly sophisticated, too, if the task of tracing the execution of a program demands it.

But developing the support for tracing is just one part of the effort of inserting tracing into a program, and, most likely, not the biggest part. The other part of the effort is calling the tracing functions at appropriate times. In large systems, this interaction with the tracing support can be overwhelming. Plus, tracing is one of those things that slows the system down, so these calls should often be pulled out of the system before the product is shipped. For these reasons, it is not unusual for developers to write ad-hoc scripting programs that rewrite the source code by inserting/deleting trace calls before and after the method bodies.

AspectJ can be used for some of these tracing concerns in a less ad-hoc way. Tracing can be seen as a concern that crosscuts the entire system and as such is amenable to encapsulation in an aspect. In addition, it is fairly independent of what the system is doing. Therefore tracing is one of those kind of system aspects that can potentially be plugged in and unplugged without any side-effects in the basic functionality of the system.

4.4.3.1.1 An Example Application

Throughout this example we will use a simple application that contains only four classes. The application is about shapes. The `TwoDShape` class is the root of the shape hierarchy:

```
public abstract class TwoDShape {
    protected double x, y;
    protected TwoDShape(double x, double y) {
        this.x = x; this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double distance(TwoDShape s) {
        double dx = Math.abs(s.getX() - x);
        double dy = Math.abs(s.getY() - y);
        return Math.sqrt(dx*dx + dy*dy);
    }
    public abstract double perimeter();
    public abstract double area();
    public String toString() {
        return (" @ (" + String.valueOf(x) + ", " + String.valueOf(y) +
            ") ");
    }
}
```

`TwoDShape` has two subclasses, `Circle` and `Square`:

```

public class Circle extends TwoDShape {
    protected double r;
    public Circle(double x, double y, double r) {
        super(x, y); this.r = r;
    }
    public Circle(double x, double y) { this( x, y, 1.0); }
    public Circle(double r)           { this(0.0, 0.0, r); }
    public Circle()                   { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 2 * Math.PI * r;
    }
    public double area() {
        return Math.PI * r*r;
    }
    public String toString() {
        return ("Circle radius = " + String.valueOf(r) +
super.toString());
    }
}

public class Square extends TwoDShape {
    protected double s; // side
    public Square(double x, double y, double s) {
        super(x, y); this.s = s;
    }
    public Square(double x, double y) { this( x, y, 1.0); }
    public Square(double s)           { this(0.0, 0.0, s); }
    public Square()                   { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 4 * s;
    }
    public double area() {
        return s*s;
    }
    public String toString() {
        return ("Square side = " + String.valueOf(s) +
super.toString());
    }
}

```

To run this application, compile the classes. You can do it with or without ajc, the AspectJ compiler. If you've installed AspectJ, go to the directory *InstallDir/examples* and type:

```
ajc -argfile tracing/notrace.lst
```

To run the program, type

```
java tracing.ExampleMain
```

(we don't need anything special on the classpath since this is pure Java code). You should see the following output:

```

c1.perimeter() = 12.566370614359172
c1.area() = 12.566370614359172
s1.perimeter() = 4.0

```

```

s1.area() = 1.0
c2.distance(c1) = 4.242640687119285
s1.distance(c1) = 2.23606797749979
s1.toString(): Square side = 1.0 @ (1.0, 2.0)

```

4.4.3.1.2 Tracing—Version 1

In a first attempt to insert tracing in this application, we will start by writing a `Trace` class that is exactly what we would write if we didn't have aspects. The implementation is in `version1/Trace.java`. Its public interface is:

```

public class Trace {
    public static int TRACELEVEL = 0;
    public static void initStream(PrintStream s) {...}
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
}

```

If we didn't have AspectJ, we would have to insert calls to `traceEntry` and `traceExit` in all methods and constructors we wanted to trace, and to initialize `TRACELEVEL` and the stream. If we wanted to trace all the methods and constructors in our example, that would amount to around 40 calls, and we would hope we had not forgotten any method. But we can do that more consistently and reliably with the following aspect (found in `version1/TraceMyClasses.java`):

```

aspect TraceMyClasses {
    pointcut myClass(): within(TwoDShape) || within(Circle) ||
within(Square);
    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before (): myMethod() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}

```

This aspect performs the tracing calls at appropriate times. According to this aspect, tracing is performed at the entrance and exit of every method and constructor defined within the shape hierarchy.

What is printed at before and after each of the traced join points is the signature of the method executing. Since the signature is static information, we can get it through `thisJoinPointStaticPart`.

To run this version of tracing, go to the directory `InstallDir/examples` and type:

```
ajc -argfile tracing/tracev1.lst
```

Running the main method of `tracing.version1.TraceMyClasses` should produce the output:

```
--> tracing.TwoDShape(double, double)
<-- tracing.TwoDShape(double, double)
--> tracing.Circle(double, double, double)
<-- tracing.Circle(double, double, double)
--> tracing.TwoDShape(double, double)
<-- tracing.TwoDShape(double, double)
--> tracing.Circle(double, double, double)
<-- tracing.Circle(double, double, double)
--> tracing.Circle(double)
<-- tracing.Circle(double)
--> tracing.TwoDShape(double, double)
<-- tracing.TwoDShape(double, double)
--> tracing.Square(double, double, double)
<-- tracing.Square(double, double, double)
--> tracing.Square(double, double)
<-- tracing.Square(double, double)
--> double tracing.Circle.perimeter()
<-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
--> double tracing.Circle.area()
<-- double tracing.Circle.area()
c1.area() = 12.566370614359172
--> double tracing.Square.perimeter()
<-- double tracing.Square.perimeter()
s1.perimeter() = 4.0
--> double tracing.Square.area()
<-- double tracing.Square.area()
s1.area() = 1.0
--> double tracing.TwoDShape.distance(TwoDShape)
--> double tracing.TwoDShape.getX()
<-- double tracing.TwoDShape.getX()
--> double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
--> double tracing.TwoDShape.distance(TwoDShape)
--> double tracing.TwoDShape.getX()
<-- double tracing.TwoDShape.getX()
--> double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
--> String tracing.Square.toString()
--> String tracing.TwoDShape.toString()
<-- String tracing.TwoDShape.toString()
<-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

When `TraceMyClasses.java` is not provided to **ajc**, the aspect does not have any affect on the system and the tracing is unplugged.

4.4.3.1.3 Tracing—Version 2

Another way to accomplish the same thing would be to write a reusable tracing aspect that can be used not only for these application classes, but for any class. One way to do this is to merge the tracing functionality of `Trace-version1` with the crosscutting support of `TraceMyClasses-version1`. We end up with a `Trace` aspect (found in `version2/Trace.java`) with the following public interface

```
abstract aspect Trace {

    public static int TRACELEVEL = 2;
    public static void initStream(PrintStream s) {...}
    protected static void traceEntry(String str) {...}
    protected static void traceExit(String str) {...}
    abstract pointcut myClass();
}
```

In order to use it, we need to define our own subclass that knows about our application classes, in `version2/TraceMyClasses.java`:

```
public aspect TraceMyClasses extends Trace {
    pointcut myClass(): within(TwoDShape) || within(Circle) ||
within(Square);

    public static void main(String[] args) {
        Trace.TRACELEVEL = 2;
        Trace.initStream(System.err);
        ExampleMain.main(args);
    }
}
```

Notice that we've simply made the pointcut concrete, that was an abstract pointcut in the super-aspect, concrete. To run this version of tracing, go to the directory `examples` and type:

```
ajc -argfile tracing/tracev2.lst
```

The file `tracev2.lst` lists the application classes as well as this version of the files `Trace.java` and `TraceMyClasses.java`. Running the main method of `tracing.version2.TraceMyClasses` should output exactly the same trace information as that from version 1.

The entire implementation of the new `Trace` class is:

```
abstract aspect Trace {

    // implementation part

    public static int TRACELEVEL = 2;
    protected static PrintStream stream = System.err;
    protected static int callDepth = 0;

    public static void initStream(PrintStream s) {
        stream = s;
    }
}
```

```

protected static void traceEntry(String str) {
    if (TRACELEVEL == 0) return;
    if (TRACELEVEL == 2) callDepth++;
    printEntering(str);
}
protected static void traceExit(String str) {
    if (TRACELEVEL == 0) return;
    printExiting(str);
    if (TRACELEVEL == 2) callDepth--;
}
private static void printEntering(String str) {
    printIndent();
    stream.println("--> " + str);
}
private static void printExiting(String str) {
    printIndent();
    stream.println("<-- " + str);
}
private static void printIndent() {
    for (int i = 0; i < callDepth; i++)
        stream.print(" ");
}

// protocol part

abstract pointcut myClass();

pointcut myConstructor(): myClass() && execution(new(..));
pointcut myMethod(): myClass() && execution(* *(..));

before(): myConstructor() {
    traceEntry("" + thisJoinPointStaticPart.getSignature());
}
after(): myConstructor() {
    traceExit("" + thisJoinPointStaticPart.getSignature());
}

before(): myMethod() {
    traceEntry("" + thisJoinPointStaticPart.getSignature());
}
after(): myMethod() {
    traceExit("" + thisJoinPointStaticPart.getSignature());
}
}

```

This version differs from version 1 in several subtle ways. The first thing to notice is that this `Trace` class merges the functional part of tracing with the crosscutting of the tracing calls. That is, in version 1, there was a sharp separation between the tracing support (the class `Trace`) and the crosscutting usage of it (by the class `TraceMyClasses`). In this version those two things are merged. That's why the description of this class explicitly says that "Trace messages are printed before and after constructors and methods are," which is what we wanted in the first place. That is, the placement of the calls, in this version, is established by the aspect class itself, leaving less opportunity for misplacing calls.

A consequence of this is that there is no need for providing `traceEntry` and `traceExit` as public operations of this class. You can see that they were classified as protected. They are supposed to be internal implementation details of the advice.

The key piece of this aspect is the abstract pointcut classes that serves as the base for the definition of the pointcuts constructors and methods. Even though `classes` is abstract, and therefore no concrete classes are mentioned, we can put advice on it, as well as on the pointcuts that are based on it. The idea is "we don't know exactly what the pointcut will be, but when we do, here's what we want to do with it." In some ways, abstract pointcuts are similar to abstract methods. Abstract methods don't provide the implementation, but you know that the concrete subclasses will, so you can invoke those methods.

4.4.4 Production Aspects

The following aspects are useful in production.

4.4.4.1 A Bean Aspect

(The code for this example is in *InstallDir/examples/bean*.)

This example examines an aspect that makes `Point` objects into Java beans with bound properties.

Java beans are reusable software components that can be visually manipulated in a builder tool. The requirements for an object to be a bean are few. Beans must define a no-argument constructor and must be either `Serializable` or `Externalizable`. Any properties of the object that are to be treated as bean properties should be indicated by the presence of appropriate `get` and `set` methods whose names are `getProperty` and `setProperty` where *property* is the name of a field in the bean class. Some bean properties, known as bound properties, fire events whenever their values change so that any registered listeners (such as, other beans) will be informed of those changes. Making a bound property involves keeping a list of registered listeners, and creating and dispatching event objects in methods that change the property values, such as `setProperty` methods.

`Point` is a simple class representing points with rectangular coordinates. `Point` does not know anything about being a bean: there are set methods for `x` and `y` but they do not fire events, and the class is not serializable. `Bound` is an aspect that makes `Point` a serializable class and makes its `get` and `set` methods support the bound property protocol.

4.4.4.1.1 The `Point` class

The `Point` class is a very simple class with trivial getters and setters, and a simple vector offset method.

```
class Point {  
    protected int x = 0;
```

```

protected int y = 0;

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public void setRectangular(int newX, int newY) {
    setX(newX);
    setY(newY);
}

public void setX(int newX) {
    x = newX;
}

public void setY(int newY) {
    y = newY;
}

public void offset(int deltaX, int deltaY) {
    setRectangular(x + deltaX, y + deltaY);
}

public String toString() {
    return "(" + getX() + ", " + getY() + ")";
}
}

```

4.4.4.1.2 *The BoundPoint aspect*

The `BoundPoint` aspect is responsible for `Point`'s "beanness". The first thing it does is privately declare that each `Point` has a support field that holds reference to an instance of `PropertyChangeSupport`.

```

private PropertyChangeSupport Point.support = new
PropertyChangeSupport(this);

```

The property change support object must be constructed with a reference to the bean for which it is providing support, so it is initialized by passing it `this`, an instance of `Point`. Since the `support` field is private declared in the aspect, only the code in the aspect can refer to it.

The aspect also declares `Point`'s methods for registering and managing listeners for property change events, which delegate the work to the property change support object:

```

public void Point.addPropertyChangeListener(PropertyChangeListener
listener) {
    support.addPropertyChangeListener(listener);
}
public void Point.addPropertyChangeListener(String propertyName,
                                           PropertyChangeListener
listener) {

```



```

        support.addPropertyChangeListener(propertyName, listener);
    }
    public void Point.removePropertyChangeListener(String propertyName,
                                                    PropertyChangeListener
listener) {
        support.removePropertyChangeListener(propertyName, listener);
    }
    public void Point.removePropertyChangeListener(PropertyChangeListener
listener) {
        support.removePropertyChangeListener(listener);
    }
    public void Point.hasListeners(String propertyName) {
        support.hasListeners(propertyName);
    }
}

```

The aspect is also responsible for making sure `Point` implements the `Serializable` interface:

```

declare parents: Point implements Serializable;

```

Implementing this interface in Java does not require any methods to be implemented.

Serialization for `Point` objects is provided by the default serialization method.

The `setters` pointcut picks out calls to the `Point`'s `set` methods: any method whose name begins with "set" and takes one parameter. The `around` advice on `setters()` stores the values of the `x` and `y` properties, calls the original `set` method and then fires the appropriate property change event according to which `set` method was called.

```

aspect BoundPoint {
    private PropertyChangeSupport Point.support = new
PropertyChangeSupport(this);

    public void Point.addPropertyChangeListener(PropertyChangeListener
listener) {
        support.addPropertyChangeListener(listener);
    }
    public void Point.addPropertyChangeListener(String propertyName,
                                                    PropertyChangeListener
listener) {
        support.addPropertyChangeListener(propertyName, listener);
    }
    public void Point.removePropertyChangeListener(String propertyName,
                                                    PropertyChangeListener
listener) {
        support.removePropertyChangeListener(propertyName, listener);
    }
    public void Point.removePropertyChangeListener(PropertyChangeListener
listener) {
        support.removePropertyChangeListener(listener);
    }
    public void Point.hasListeners(String propertyName) {
        support.hasListeners(propertyName);
    }
}

declare parents: Point implements Serializable;

```

```

pointcut setter(Point p): call(void Point.set*(*)) && target(p);

void around(Point p): setter(p) {
    String propertyName =

thisJoinPointStaticPart.getSignature().getName().substring("set".length
());
    int oldX = p.getX();
    int oldY = p.getY();
    proceed(p);
    if (propertyName.equals("X")){
        firePropertyChange(p, propertyName, oldX, p.getX());
    } else {
        firePropertyChange(p, propertyName, oldY, p.getY());
    }
}

void firePropertyChange(Point p,
                        String property,
                        double oldval,
                        double newval) {
    p.support.firePropertyChange(property,
                                new Double(oldval),
                                new Double(newval));
}
}

```

4.4.4.1.3 The Test Program

The test program registers itself as a property change listener to a `Point` object that it creates and then performs simple manipulation of that point: calling its set methods and the offset method. Then it serializes the point and writes it to a file and then reads it back. The result of saving and restoring the point is that a new point is created.

```

class Demo implements PropertyChangeListener {

    static final String fileName = "test.tmp";

    public void propertyChange(PropertyChangeEvent e){
        System.out.println("Property " + e.getPropertyName() + " changed
from " +
            e.getOldValue() + " to " + e.getNewValue() );
    }

    public static void main(String[] args){
        Point p1 = new Point();
        p1.addPropertyChangeListener(new Demo());
        System.out.println("p1 =" + p1);
        p1.setRectangular(5,2);
        System.out.println("p1 =" + p1);
        p1.setX( 6 );
        p1.setY( 3 );
        System.out.println("p1 =" + p1);
        p1.offset(6,4);
        System.out.println("p1 =" + p1);
    }
}

```

```

        save(p1, fileName);
        Point p2 = (Point) restore(fileName);
        System.out.println("Had: " + p1);
        System.out.println("Got: " + p2);
    }
    ...
}

```

4.4.4.1.4 *Compiling and Running the Example*

To compile and run this example, go to the examples directory and type:

```

ajc -argfile bean/files.lst
java bean.Demo

```

4.4.4.2 The Subject/Observer Protocol

(The code for this example is in *InstallDir/examples/observer*.)

This demo illustrates how the Subject/Observer design pattern can be coded with aspects.

The demo consists of the following: A colored label is a renderable object that has a color that cycles through a set of colors, and a number that records the number of cycles it has been through. A button is an action item that records when it is clicked.

With these two kinds of objects, we can build up a Subject/Observer relationship in which colored labels observe the clicks of buttons; that is, where colored labels are the observers and buttons are the subjects.

The demo is designed and implemented using the Subject/Observer design pattern. The remainder of this example explains the classes and aspects of this demo, and tells you how to run it.

4.4.4.2.1 *Generic Components*

The generic parts of the protocol are the interfaces `Subject` and `Observer`, and the abstract aspect `SubjectObserverProtocol`. The `Subject` interface is simple, containing methods to add, remove, and view `Observer` objects, and a method for getting data about state changes:

```

interface Subject {
    void addObserver(Observer obs);
    void removeObserver(Observer obs);
    Vector getObservers();
    Object getData();
}

```

The `Observer` interface is just as simple, with methods to set and get `Subject` objects, and a method to call when the subject gets updated.

```

interface Observer {
    void setSubject(Subject s);
    Subject getSubject();
    void update();
}

```

The `SubjectObserverProtocol` aspect contains within it all of the generic parts of the protocol, namely, how to fire the `Observer` objects' update methods when some state changes in a subject.

```
abstract aspect SubjectObserverProtocol {

    abstract pointcut stateChanges(Subject s);

    after(Subject s): stateChanges(s) {
        for (int i = 0; i < s.getObservers().size(); i++) {
            ((Observer)s.getObservers().elementAt(i)).update();
        }
    }

    private Vector Subject.observers = new Vector();
    public void Subject.addObserver(Observer obs) {
        observers.addElement(obs);
        obs.setSubject(this);
    }
    public void Subject.removeObserver(Observer obs) {
        observers.removeElement(obs);
        obs.setSubject(null);
    }
    public Vector Subject.getObservers() { return observers; }

    private Subject Observer.subject = null;
    public void Observer.setSubject(Subject s) { subject = s; }
    public Subject Observer.getSubject() { return subject; }

}
```

Note that this aspect does three things. It define an abstract pointcut that extending aspects can override. It defines advice that should run after the join points of the pointcut. And it declares an inter-type field and two inter-type methods so that each `Observer` can hold onto its `Subject`.

4.4.4.2.2 *Application Classes*

`Button` objects extend `java.awt.Button`, and all they do is make sure the `void click()` method is called whenever a button is clicked.

```
class Button extends java.awt.Button {

    static final Color defaultBackgroundColor = Color.gray;
    static final Color defaultForegroundColor = Color.black;
    static final String defaultText = "cycle color";

    Button(Display display) {
        super();
        setLabel(defaultText);
        setBackground(defaultBackgroundColor);
        setForeground(defaultForegroundColor);
        addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Button.this.click();
            }
        });
    }
}
```

```

        }
    });
    display.addToFrame(this);
}

public void click() {}
}

```

Note that this class knows nothing about being a Subject.

ColorLabel objects are labels that support the void colorCycle() method. Again, they know nothing about being an observer.

```

class ColorLabel extends Label {

    ColorLabel(Display display) {
        super();
        display.addToFrame(this);
    }

    final static Color[] colors = {Color.red, Color.blue,
                                    Color.green, Color.magenta};

    private int colorIndex = 0;
    private int cycleCount = 0;
    void colorCycle() {
        cycleCount++;
        colorIndex = (colorIndex + 1) % colors.length;
        setBackground(colors[colorIndex]);
        setText("" + cycleCount);
    }
}

```

Finally, the SubjectObserverProtocolImpl implements the subject/observer protocol, with Button objects as subjects and ColorLabel objects as observers:

```

package observer;

import java.util.Vector;

aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {

    declare parents: Button implements Subject;
    public Object Button.getData() { return this; }

    declare parents: ColorLabel implements Observer;
    public void ColorLabel.update() {
        colorCycle();
    }

    pointcut stateChanges(Subject s):
        target(s) &&
        call(void Button.click());
}

```

It does this by assuring that `Button` and `ColorLabel` implement the appropriate interfaces, declaring that they implement the methods required by those interfaces, and providing a definition for the abstract `stateChanges` pointcut. Now, every time a `Button` is clicked, all `ColorLabel` objects observing that button will `colorCycle`.

4.4.4.2.3 Compiling and Running

`Demo` is the top class that starts this demo. It instantiates a two buttons and three observers and links them together as subjects and observers. So to run the demo, go to the `examples` directory and type:

```
ajc -argfile observer/files.lst
java observer.Demo
```

4.4.4.3 A Simple Telecom Simulation

(The code for this example is in `InstallDir/examples/telecom`.)

This example illustrates some ways that dependent concerns can be encoded with aspects. It uses an example system comprising a simple model of telephone connections to which timing and billing features are added using aspects, where the billing feature depends upon the timing feature.

4.4.4.3.1 The Application

The example application is a simple simulation of a telephony system in which customers make, accept, merge and hang-up both local and long distance calls. The application architecture is in three layers.

The basic objects provide basic functionality to simulate customers, calls and connections (regular calls have one connection, conference calls have more than one).

The timing feature is concerned with timing the connections and keeping the total connection time per customer. Aspects are used to add a timer to each connection and to manage the total time per customer.

The billing feature is concerned with charging customers for the calls they make. Aspects are used to calculate a charge per connection and, upon termination of a connection, to add the charge to the appropriate customer's bill. The billing aspect builds upon the timing aspect: it uses a pointcut defined in Timing and it uses the timers that are associated with connections.

The simulation of system has three configurations: basic, timing and billing. Programs for the three configurations are in classes `BasicSimulation`, `TimingSimulation` and `BillingSimulation`. These share a common superclass `AbstractSimulation`, which defines the method `run` with the simulation itself and the method `wait` used to simulate elapsed time.

4.4.4.3.2 The Basic Objects

The telecom simulation comprises the classes `Customer`, `Call` and the abstract class `Connection` with its two concrete subclasses `Local` and `LongDistance`. Customers have

a name and a numeric area code. They also have methods for managing calls. Simple calls are made between one customer (the caller) and another (the receiver), a `Connection` object is used to connect them. Conference calls between more than two customers will involve more than one connection. A customer may be involved in many calls at one time.

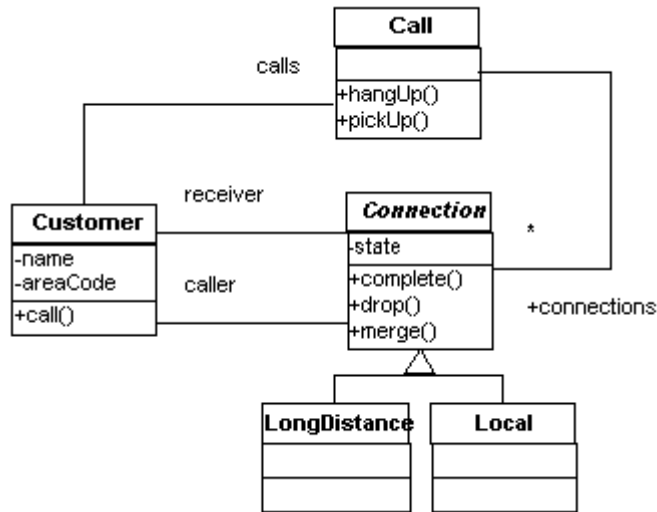


Figure 12 Telecom interactions

4.4.4.3.3 The *Customer* class

Customer has methods `call`, `pickup`, `hangup` and `merge` for managing calls.

```

public class Customer {

    private String name;
    private int areacode;
    private Vector calls = new Vector();

    protected void removeCall(Call c){
        calls.removeElement(c);
    }

    protected void addCall(Call c){
        calls.addElement(c);
    }

    public Customer(String name, int areacode) {
        this.name = name;
        this.areacode = areacode;
    }

    public String toString() {
        return name + "(" + areacode + ")";
    }

    public int getAreacode(){
        return areacode;
    }
}
  
```

```

    public boolean localTo(Customer other){
        return areacode == other.areacode;
    }

    public Call call(Customer receiver) {
        Call call = new Call(this, receiver);
        addCall(call);
        return call;
    }

    public void pickup(Call call) {
        call.pickup();
        addCall(call);
    }

    public void hangup(Call call) {
        call.hangup(this);
        removeCall(call);
    }

    public void merge(Call call1, Call call2){
        call1.merge(call2);
        removeCall(call2);
    }
}

```

4.4.4.3.4 The *call* class

Calls are created with a caller and receiver who are customers. If the caller and receiver have the same area code then the call can be established with a `Local` connection (see below), otherwise a `LongDistance` connection is required. A call comprises a number of connections between customers. Initially there is only the connection between the caller and receiver but additional connections can be added if calls are merged to form conference calls.

4.4.4.3.5 The *Connection* class

The class `Connection` models the physical details of establishing a connection between customers. It does this with a simple state machine (connections are initially `PENDING`, then `COMPLETED` and finally `DROPPED`). Messages are printed to the console so that the state of connections can be observed. `Connection` is an abstract class with two concrete subclasses: `Local` and `LongDistance`.

```

abstract class Connection {

    public static final int PENDING = 0;
    public static final int COMPLETE = 1;
    public static final int DROPPED = 2;

    Customer caller, receiver;
    private int state = PENDING;

    Connection(Customer a, Customer b) {

```



```

        this.caller = a;
        this.receiver = b;
    }

    public int getState(){
        return state;
    }

    public Customer getCaller() { return caller; }

    public Customer getReceiver() { return receiver; }

    void complete() {
        state = COMPLETE;
        System.out.println("connection completed");
    }

    void drop() {
        state = DROPPED;
        System.out.println("connection dropped");
    }

    public boolean connects(Customer c){
        return (caller == c || receiver == c);
    }
}

```

4.4.4.3.6 The *Local* and *LongDistance* classes

The two kinds of connections supported by our simulation are *Local* and *LongDistance* connections.

```

class Local extends Connection {
    Local(Customer a, Customer b) {
        super(a, b);
        System.out.println("[new local connection from " +
            a + " to " + b + "]\n");
    }
}

class LongDistance extends Connection {
    LongDistance(Customer a, Customer b) {
        super(a, b);
        System.out.println("[new long distance connection from " +
            a + " to " + b + "]\n");
    }
}

```

4.4.4.3.7 Compiling and Running the Basic Simulation

The source files for the basic system are listed in the file `basic.lst`. To build and run the basic system, in a shell window, type these commands:

```
ajc -argfile telecom/basic.lst
```

```
java telecom.BasicSimulation
```

4.4.4.3.8 The Timing aspect

The `Timing` aspect keeps track of total connection time for each `Customer` by starting and stopping a timer associated with each connection. It uses some helper classes:

4.4.4.3.8.1 The Timer class

A `Timer` object simply records the current time when it is started and stopped, and returns their difference when asked for the elapsed time. The aspect `TimerLog` (below) can be used to cause the start and stop times to be printed to standard output.

```
class Timer {
    long startTime, stopTime;

    public void start() {
        startTime = System.currentTimeMillis();
        stopTime = startTime;
    }

    public void stop() {
        stopTime = System.currentTimeMillis();
    }

    public long getTime() {
        return stopTime - startTime;
    }
}
```

4.4.4.3.9 The timerLog aspect

The `TimerLog` aspect can be included in a build to get the timer to announce when it is started and stopped.

```
public aspect TimerLog {

    after(Timer t): target(t) && call(* Timer.start()) {
        System.err.println("Timer started: " + t.startTime);
    }

    after(Timer t): target(t) && call(* Timer.stop()) {
        System.err.println("Timer stopped: " + t.stopTime);
    }
}
```

4.4.4.3.10 The Timing aspect

The `Timing` aspect is declares an inter-type field `totalConnectTime` for `Customer` to store the accumulated connection time per `Customer`. It also declares that each `Connection` object has a timer.

```
public long Customer.totalConnectTime = 0;
private Timer Connection.timer = new Timer();
```

Two pieces of after advice ensure that the timer is started when a connection is completed and stopped when it is dropped. The pointcut `endTimeing` is defined so that it can be used by the `Billing` aspect.

```
public aspect Timing {

    public long Customer.totalConnectTime = 0;

    public long getTotalConnectTime(Customer cust) {
        return cust.totalConnectTime;
    }
    private Timer Connection.timer = new Timer();
    public Timer getTimer(Connection conn) { return conn.timer; }

    after (Connection c): target(c) && call(void Connection.complete())
    {
        getTimer(c).start();
    }

    pointcut endTimeing(Connection c): target(c) &&
        call(void Connection.drop());

    after(Connection c): endTimeing(c) {
        getTimer(c).stop();
        c.getCaller().totalConnectTime += getTimer(c).getTime();
        c.getReceiver().totalConnectTime += getTimer(c).getTime();
    }
}
```

4.4.4.3.11 *The Billing aspect*

The `Billing` system adds billing functionality to the telecom application on top of timing.

The `Billing` aspect declares that each `Connection` has a `payer` inter-type field to indicate who initiated the call and therefore who is responsible to pay for it. It also declares the inter-type method `callRate` of `Connection` so that local and long distance calls can be charged differently. The call charge must be calculated after the timer is stopped; the after advice on pointcut `Timing.endTimeing` does this, and `Billing` is declared to be more precedent than `Timing` to make sure that this advice runs after `Timing`'s advice on the same join point. Finally, it declares inter-type methods and fields for `Customer` to handle the `totalCharge`.

```
public aspect Billing {
    // precedence required to get advice on endtiming in the right
    order
    declare precedence: Billing, Timing;

    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;

    public Customer Connection.payer;
    public Customer getPayer(Connection conn) { return conn.payer; }

    after(Customer cust) returning (Connection conn):
```

```

        args(cust, ..) && call(Connection+.new(..)) {
            conn.payer = cust;
        }

public abstract long Connection.callRate();

public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
public long Local.callRate() { return LOCAL_RATE; }

after(Connection conn): Timing.endTiming(conn) {
    long time = Timing.aspectOf().getTimer(conn).getTime();
    long rate = conn.callRate();
    long cost = rate * time;
    getPayer(conn).addCharge(cost);
}

public long Customer.totalCharge = 0;
public long getTotalCharge(Customer cust) { return
cust.totalCharge; }

public void Customer.addCharge(long charge){
    totalCharge += charge;
}
}

```

4.4.4.3.12 *Accessing the inter-type state*

Both the aspects `Timing` and `Billing` contain the definition of operations that the rest of the system may want to access. For example, when running the simulation with one or both aspects, we want to find out how much time each customer spent on the telephone and how big their bill is. That information is also stored in the classes, but they are accessed through static methods of the aspects, since the state they refer to is private to the aspect.

Take a look at the file `TimingSimulation.java`. The most important method of this class is the method `report(Customer)`, which is used in the method `run` of the superclass `AbstractSimulation`. This method is intended to print out the status of the customer, with respect to the `Timing` feature.

```

protected void report(Customer c){
    Timing t = Timing.aspectOf();
    System.out.println(c + " spent " + t.getTotalConnectTime(c));
}

```

4.4.4.3.13 *Compiling and Running*

The files `timing.lst` and `billing.lst` contain file lists for the timing and billing configurations. To build and run the application with only the timing feature, go to the directory `examples` and type:

```

ajc -argfile telecom/timing.lst
java telecom.TimingSimulation

```

To build and run the application with the timing and billing features, go to the directory `examples` and type:

```
ajc -argfile telecom/billing.lst
java telecom.BillingSimulation
```

4.4.4.3.14 Discussion

There are some explicit dependencies between the aspects `Billing` and `Timing`:

`Billing` is declared more precedent than `Timing` so that `Billing`'s after advice runs after that of `Timing` when they are on the same join point.

`Billing` uses the pointcut `Timing.endTiming`.

`Billing` needs access to the timer associated with a connection.

4.4.5 Reusable Aspects

The following aspects are reusable.

4.4.5.1 Tracing using Aspects, Revisited

(The code for this example is in `InstallDir/examples/tracing`.)

4.4.5.1.1 Tracing—Version 3

One advantage of not exposing the methods `traceEntry` and `traceExit` as public operations is that we can easily change their interface without any dramatic consequences in the rest of the code.

Consider, again, the program without `AspectJ`. Suppose, for example, that at some point later the requirements for tracing change, stating that the trace messages should always include the string representation of the object whose methods are being traced. This can be achieved in at least two ways. One way is keep the interface of the methods `traceEntry` and `traceExit` as it was before,

```
public static void traceEntry(String str);
public static void traceExit(String str);
```

In this case, the caller is responsible for ensuring that the string representation of the object is part of the string given as argument. So, calls must look like:

```
Trace.traceEntry("Square.distance in " + toString());
```

Another way is to enforce the requirement with a second argument in the trace operations, e.g.

```
public static void traceEntry(String str, Object obj);
public static void traceExit(String str, Object obj);
```

In this case, the caller is still responsible for sending the right object, but at least there is some guarantees that some object will be passed. The calls will look like:

```
Trace.traceEntry("Square.distance", this);
```

In either case, this change to the requirements of tracing will have dramatic consequences in the rest of the code -- every call to the trace operations `traceEntry` and `traceExit` must be changed!

Here's another advantage of doing tracing with an aspect. We've already seen that in version 2 `traceEntry` and `traceExit` are not publicly exposed. So changing their interfaces, or the way they are used, has only a small effect inside the `Trace` class. Here's a partial view at the implementation of `Trace`, version 3. The differences with respect to version 2 are stressed in the comments:

```
abstract aspect Trace {

    public static int TRACELEVEL = 0;
    protected static PrintStream stream = null;
    protected static int callDepth = 0;

    public static void initStream(PrintStream s) {
        stream = s;
    }

    protected static void traceEntry(String str, Object o) {
        if (TRACELEVEL == 0) return;
        if (TRACELEVEL == 2) callDepth++;
        printEntering(str + ": " + o.toString());
    }

    protected static void traceExit(String str, Object o) {
        if (TRACELEVEL == 0) return;
        printExiting(str + ": " + o.toString());
        if (TRACELEVEL == 2) callDepth--;
    }

    private static void printEntering(String str) {
        printIndent();
        stream.println("Entering " + str);
    }

    private static void printExiting(String str) {
        printIndent();
        stream.println("Exiting " + str);
    }

    private static void printIndent() {
        for (int i = 0; i < callDepth; i++)
            stream.print(" ");
    }

    abstract pointcut myClass(Object obj);

    pointcut myConstructor(Object obj): myClass(obj) &&
execution(new(..));
    pointcut myMethod(Object obj): myClass(obj) &&
        execution(* *(..)) && !execution(String toString());
```

```

before(Object obj): myConstructor(obj) {
    traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
}
after(Object obj): myConstructor(obj) {
    traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
}

before(Object obj): myMethod(obj) {
    traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
}
after(Object obj): myMethod(obj) {
    traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
}
}

```

As you can see, we decided to apply the first design by preserving the interface of the methods `traceEntry` and `traceExit`. But it doesn't matter—we could as easily have applied the second design (the code in the directory `examples/tracing/version3` has the second design). The point is that the effects of this change in the tracing requirements are limited to the `Trace` aspect class.

One implementation change worth noticing is the specification of the pointcuts. They now expose the object. To maintain full consistency with the behavior of version 2, we should have included tracing for static methods, by defining another pointcut for static methods and advising it. We leave that as an exercise.

Moreover, we had to exclude the execution join point of the method `toString` from the `methods` pointcut. The problem here is that `toString` is being called from inside the advice. Therefore if we trace it, we will end up in an infinite recursion of calls. This is a subtle point, and one that you must be aware when writing advice. If the advice calls back to the objects, there is always the possibility of recursion. Keep that in mind!

In fact, simply excluding the execution join point may not be enough, if there are calls to other traced methods within it -- in which case, the restriction should be

```

&& !cflow(execution(String toString()))

```

excluding both the execution of `toString` methods and all join points under that execution.

In summary, to implement the change in the tracing requirements we had to make a couple of changes in the implementation of the `Trace` aspect class, including changing the specification of the pointcuts. That's only natural. But the implementation changes were limited to this aspect. Without aspects, we would have to change the implementation of every application class.

Finally, to run this version of tracing, go to the directory `examples` and type:

```

ajc -argfile tracing/tracev3.lst

```

The file `tracev3.lst` lists the application classes as well as this version of the files `Trace.java` and `TraceMyClasses.java`. To run the program, type

```

java tracing.version3.TraceMyClasses

```

The output should be:

```

--> tracing.TwoDShape(double, double)
<-- tracing.TwoDShape(double, double)
--> tracing.Circle(double, double, double)
<-- tracing.Circle(double, double, double)
--> tracing.TwoDShape(double, double)
<-- tracing.TwoDShape(double, double)
--> tracing.Circle(double, double, double)
<-- tracing.Circle(double, double, double)
--> tracing.Circle(double)
<-- tracing.Circle(double)
--> tracing.TwoDShape(double, double)
<-- tracing.TwoDShape(double, double)
--> tracing.Square(double, double, double)
<-- tracing.Square(double, double, double)
--> tracing.Square(double, double)
<-- tracing.Square(double, double)
--> double tracing.Circle.perimeter()
<-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
--> double tracing.Circle.area()
<-- double tracing.Circle.area()
c1.area() = 12.566370614359172
--> double tracing.Square.perimeter()
<-- double tracing.Square.perimeter()
s1.perimeter() = 4.0
--> double tracing.Square.area()
<-- double tracing.Square.area()
s1.area() = 1.0
--> double tracing.TwoDShape.distance(TwoDShape)
--> double tracing.TwoDShape.getX()
<-- double tracing.TwoDShape.getX()
--> double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
--> double tracing.TwoDShape.distance(TwoDShape)
--> double tracing.TwoDShape.getX()
<-- double tracing.TwoDShape.getX()
--> double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.getY()
<-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
--> String tracing.Square.toString()
--> String tracing.TwoDShape.toString()
<-- String tracing.TwoDShape.toString()
<-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)

```

4.5 Idioms

This chapter consists of very short snippets of AspectJ code, typically pointcuts, that are particularly evocative or useful. This section is a work in progress.

Here's an example of how to enforce a rule that code in the `java.sql` package can only be used from one particular package in your system. This doesn't require any access to code in the `java.sql` package.

```
/* Any call to methods or constructors in java.sql */
pointcut restrictedCall():
    call(* java.sql.*(..)) || call(java.sql.*.new(..));

/* Any code in my system not in the sqlAccess package */
pointcut illegalSource():
    within(com.foo..*) && !within(com.foo.sqlAccess.*);

declare error: restrictedCall() && illegalSource():
    "java.sql package can only be accessed from com.foo.sqlAccess";
```

Any call to an instance of a subtype of `AbstractFacade` whose class is not exactly equal to `AbstractFacade`:

```
pointcut nonAbstract(AbstractFacade af):
    call(* *(..))
    && target(af)
    && !if(af.getClass() == AbstractFacade.class);
```

If `AbstractFacade` is an abstract class or an interface, then every instance must be of a subtype and you can replace this with:

```
pointcut nonAbstract(AbstractFacade af):
    call(* *(..))
    && target(af);
```

Any call to a method which is defined by a subtype of `AbstractFacade`, but which isn't defined by the type `AbstractFacade` itself:

```
pointcut callToUndefinedMethod():
    call(* AbstractFacade+.*(..))
    && !call(* AbstractFacade.*(..));
```

The execution of a method that is defined in the source code for a type that is a subtype of `AbstractFacade` but not in `AbstractFacade` itself:

```
pointcut executionOfUndefinedMethod():
    execution(* *(..))
    && within(AbstractFacade+)
    && !within(AbstractFacade)
```

4.6 Pitfalls

This chapter consists of a few AspectJ programs that may lead to surprising behavior and how to understand them.

4.6.1 Infinite loops

Here is a Java program with peculiar behavior

```

public class Main {
    public static void main(String[] args) {
        foo();
        System.out.println("done with call to foo");
    }

    static void foo() {
        try {
            foo();
        } finally {
            foo();
        }
    }
}

```

This program will never reach the println call, but when it aborts may have no stack trace.

This silence is caused by multiple StackOverflowExceptions. First the infinite loop in the body of the method generates one, which the finally clause tries to handle. But this finally clause also generates an infinite loop which the current JVMs can't handle gracefully leading to the completely silent abort.

The following short aspect will also generate this behavior:

```

aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after(): call(* *(..)) { System.out.println("after"); }
}

```

Why? Because the call to println is also a call matched by the pointcut call (* *(..)). We get no output because we used simple after() advice. If the aspect were changed to

```

aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after() returning: call(* *(..)) { System.out.println("after"); }
}

```

Then at least a StackOverflowException with a stack trace would be seen. In both cases, though, the overall problem is advice applying within its own body.

There's a simple idiom to use if you ever have a worry that your advice might apply in this way. Just restrict the advice from occurring in join points caused within the aspect. So:

```

aspect A {
    before(): call(* *(..)) && !within(A) {
        System.out.println("before"); }
    after() returning: call(* *(..)) && !within(A) {
        System.out.println("after"); }
}

```

Other solutions might be to more closely restrict the pointcut in other ways, for example:

```

aspect A {
    before(): call(* MyObject.*(..)) { System.out.println("before"); }
}

```

```
    after() returning: call(* MyObject.*(..)) {  
System.out.println("after"); }  
}
```

The moral of the story is that unrestricted generic pointcuts can pick out more join points than intended.

Appendix A AspectJ Quick Reference

Table of Contents

[Pointcuts](#)

[Type Patterns](#)

[Advice](#)

[Inter-type member declarations](#)

[Other declarations](#)

[Aspects](#)

A.1 Pointcuts

Methods and Constructors

`call(Signature)`

every call to any method or constructor
matching *Signature* at the call site

`execution(Signature)`

every execution of any method or constructor
matching *Signature*

Fields

`get(Signature)`

every reference to any field matching
Signature

`set(Signature)`

every assignment to any field matching
Signature. The assigned value can be exposed
with an `args` pointcut

Exception Handlers

`handler(TypePattern)`

every exception handler for any `Throwable`
type in *TypePattern*. The exception value can
be exposed with an `args` pointcut

Advice

`adviceexecution()`

every execution of any piece of advice

Initialization

`staticinitialization(TypePattern)`

every execution of a static initializer for any
type in *TypePattern*

`initialization(Signature)`

every initialization of an object when the first
constructor called in the type matches
Signature, encompassing the return from the
super constructor call to the return of the first-
called constructor

`preinitialization(Signature)`

every pre-initialization of an object when the
first constructor called in the type matches
Signature, encompassing the entry of the first-
called constructor to the call to the super
constructor

Lexical

<code>within(<i>TypePattern</i>)</code>	every join point from code defined in a type in <i>TypePattern</i>
<code>withincode(<i>Signature</i>)</code>	every join point from code defined in a method or constructor matching <i>Signature</i>

Instanceof checks and context exposure

<code>this(<i>Type</i> or <i>Id</i>)</code>	every join point when the currently executing object is an instance of <i>Type</i> or <i>Id</i> 's type
<code>target(<i>Type</i> or <i>Id</i>)</code>	every join point when the target executing object is an instance of <i>Type</i> or <i>Id</i> 's type
<code>args(<i>Type</i> or <i>Id</i>, ...)</code>	every join point when the arguments are instances of <i>Types</i> or the types of the <i>Ids</i>

Control Flow

<code>cflow(<i>Pointcut</i>)</code>	every join point in the control flow of each join point <i>P</i> picked out by <i>Pointcut</i> , including <i>P</i> itself
<code>cflowbelow(<i>Pointcut</i>)</code>	every join point below the control flow of each join point <i>P</i> picked out by <i>Pointcut</i> ; does not include <i>P</i> itself

Conditional

<code>if(<i>Expression</i>)</code>	every join point when the boolean <i>Expression</i> is true
------------------------------------	---

Combination

<code>! <i>Pointcut</i></code>	every join point not picked out by <i>Pointcut</i>
<code><i>Pointcut0</i> && <i>Pointcut1</i></code>	each join point picked out by both <i>Pointcut0</i> and <i>Pointcut1</i>
<code><i>Pointcut0</i> <i>Pointcut1</i></code>	each join point picked out by either <i>Pointcut0</i> or <i>Pointcut1</i>
<code>(<i>Pointcut</i>)</code>	each join point picked out by <i>Pointcut</i>

A.2 Type Patterns

A type pattern is one of

<code><i>TypeNamePattern</i></code>	all types in <i>TypeNamePattern</i>
<code><i>SubtypePattern</i></code>	all types in <i>SubtypePattern</i> , a pattern with a +.
<code><i>ArrayTypePattern</i></code>	all types in <i>ArrayTypePattern</i> , a pattern with one or more [].
<code>! <i>TypePattern</i></code>	all types not in <i>TypePattern</i>
<code><i>TypePattern0</i> && <i>TypePattern1</i></code>	all types in both <i>TypePattern0</i> and <i>TypePattern1</i>
<code><i>TypePattern0</i> <i>TypePattern1</i></code>	all types in either <i>TypePattern0</i> or <i>TypePattern1</i>
<code>(<i>TypePattern</i>)</code>	all types in <i>TypePattern</i>

where *TypeNamePattern* can either be a plain type name, the wildcard * (indicating all types), or an identifier with embedded * and .. wildcards.

An embedded * in an identifier matches any sequence of characters, but does not match the package (or inner-type) separator ".".

An embedded .. in an identifier matches any sequence of characters that starts and ends with the package (or inner-type) separator ".".

A.3 Advice

Each piece of advice is of the form

```
[ strictfp ] AdviceSpec [ throws TypeList ] : Pointcut {  
    Body }
```

where *AdviceSpec* is one of

before(Formals)

runs before each join point

after(Formals) returning [(Formal)]

runs after each join point that returns normally. The optional formal gives access to the returned value

after(Formals) throwing [(Formal)]

runs after each join point that throws a *Throwable*. If the optional formal is present, runs only after each join point that throws a *Throwable* of the type of *Formal*, and *Formal* gives access to the *Throwable* exception value

after(Formals)

runs after each join point regardless of whether it returns normally or throws a *Throwable*

Type around(Formals)

runs in place of each join point. The join point can be executed by calling *proceed*, which takes the same number and types of arguments as the *around* advice.

Three special variables are available inside of advice bodies:

thisJoinPoint

an object of type org.aspectj.lang.JoinPoint representing the join point at which the advice is executing.

thisJoinPointStaticPart

equivalent to *thisJoinPoint.getStaticPart()*, but may use fewer runtime resources.

thisEnclosingJoinPointStaticPart

the static part of the dynamically enclosing join point.

A.4 Inter-type member declarations

Each inter-type member is one of

```
Modifiers ReturnType OnType . Id ( Formals ) [ throws TypeList ] { Body  
}
```

a method on *OnType*.

```

abstract Modifiers ReturnType OnType . Id ( Formals ) [ throws TypeList
] ;
    an abstract method on OnType.
Modifiers OnType . new ( Formals ) [ throws TypeList ] { Body }
    a constructor on OnType.
Modifiers Type OnType . Id [ = Expression ] ;
    a field on OnType.

```

A.5 Other declarations

```

declare parents : TypePattern extends Type ;
    the types in TypePattern extend Type.
declare parents : TypePattern implements TypeList ;
    the types in TypePattern implement the types in TypeList.
declare warning : Pointcut : String ;
    if any of the join points in Pointcut possibly exist in the program, the compiler
    emits the warning String.
declare error : Pointcut : String ;
    if any of the join points in Pointcut could possibly exist in the program, the
    compiler emits the error String.
declare soft : Type : Pointcut ;
    any Type exception that gets thrown at any join point picked out by Pointcut is
    wrapped in org.aspectj.lang.SoftException.
declare precedence : TypePatternList ;
    at any join point where multiple pieces of advice apply, the advice precedence at
    that join point is in TypePatternList order.

```

A.6 Aspects

Each aspect is of the form

```

[ privileged ] Modifiers aspect Id [ extends Type ] [
implements TypeList ] [ PerClause ] { Body }

```

where *PerClause* defines how the aspect is instantiated and associated (issingleton by default):

PerClause	Description	Accessor
[issingleton]	One instance of the aspect is made. This is the default.	aspectOf () at all join points
perthis (Pointcut)	An instance is associated with each object that is the currently executing object at any join point in Pointcut.	aspectOf (Object) at all join points
pertarget (Pointcut)	An instance is associated with each object that is the target object at any join point in Pointcut.	aspectOf (Object) at all join points

PerClause	Description	Accessor
<code>percfow(<i>Pointcut</i>)</code>	The aspect is defined for each entrance to the control flow of the join points defined by <i>Pointcut</i> .	<code>aspectOf()</code> at join points in <code>cflow(<i>Pointcut</i>)</code>
<code>percfowbelow(<i>Pointcut</i>)</code>	The aspect is defined for each entrance to the control flow below the join points defined by <i>Pointcut</i> .	<code>aspectOf()</code> at join points in <code>cflowbelow(<i>Pointcut</i>)</code>

Appendix B Language Semantics

Table of Contents

[Introduction](#)

[Join Points](#)

[Pointcuts](#)

[Pointcut definition](#)

[Context exposure](#)

[Primitive pointcuts](#)

[Signatures](#)

[Matching](#)

[Type patterns](#)

[Advice](#)

[Advice modifiers](#)

[Advice and checked exceptions](#)

[Advice precedence](#)

[Reflective access to the join point](#)

[Static crosscutting](#)

[Inter-type member declarations](#)

[Access modifiers](#)

[Conflicts](#)

[Extension and Implementation](#)

[Interfaces with members](#)

[Warnings and Errors](#)

[Softened exceptions](#)

[Advice Precedence](#)

[Statically determinable pointcuts](#)

[Aspects](#)

[Aspect Extension](#)

[Aspect instantiation](#)

[Aspect privilege](#)

B.1 Introduction

AspectJ extends Java by overlaying a concept of join points onto the existing Java semantics and adding a few new program elements to Java:

A join point is a well-defined point in the execution of a program. These include method and constructor calls, field accesses and others described below.

A pointcut picks out join points, and exposes some of the values in the execution context of those join points. There are several primitive pointcut designators, and others can be named and defined by the `pointcut` declaration.

A piece of advice is code that executes at each join point in a pointcut. Advice has access to the values exposed by the pointcut. Advice is defined by `before`, `after`, and `around` declarations.

Inter-type declarations form AspectJ's static crosscutting features, that is, is code that may change the type structure of a program, by adding to or extending interfaces and classes with new fields, constructors, or methods. Some inter-type declarations are defined through an extension of usual method, field, and constructor declarations, and other declarations are made with a new `declare` keyword.

An aspect is a crosscutting type that encapsulates pointcuts, advice, and static crosscutting features. By type, we mean Java's notion: a modular unit of code, with a well-defined interface, about which it is possible to do reasoning at compile time. Aspects are defined by the `aspect` declaration.

B.2 Join Points

While aspects define types that crosscut, the AspectJ system does not allow completely arbitrary crosscutting. Rather, aspects define types that cut across principled points in a program's execution. These principled points are called join points.

A join point is a well-defined point in the execution of a program. The join points defined by AspectJ are:

Method call

When a method is called, not including super calls of non-static methods.

Method execution

When the body of code for an actual method executes.

Constructor call

When an object is built and that object's initial constructor is called (i.e., not for "super" or "this" constructor calls). The object being constructed is returned at a constructor call join point, so its return type is considered to be the type of the object, and the object itself may be accessed with `after returning` advice.

Constructor execution

When the body of code for an actual constructor executes, after its this or super constructor call. The object being constructed is the currently executing object, and so may be accessed with the `this` pointcut. The constructor execution join point for a constructor that calls a super constructor also includes any non-static initializers of enclosing class. No value is returned from a constructor execution join point, so its return type is considered to be void.

Static initializer execution

When the static initializer for a class executes. No value is returned from a static initializer execution join point, so its return type is considered to be void.

Object pre-initialization

Before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. Thus, the execution of these join points encompass the join points of the evaluation of the arguments of `this()` and `super()` constructor calls. No value is returned from an object pre-initialization join point, so its return type is considered to be void.

Object initialization

When the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object. The object being constructed is the currently executing object, and so may be accessed with the `this` pointcut. No value is returned from a constructor execution join point, so its return type is considered to be void.

Field reference

When a non-constant field is referenced. [Note that references to constant fields (static final fields bound to a constant string object or primitive value) are not join points, since Java requires them to be inlined.]

Field set

When a field is assigned to. Field set join points are considered to have one argument, the value the field is being set to. No value is returned from a field set join point, so its return type is considered to be void. [Note that the initializations of constant fields (static final fields where the initializer is a constant string object or primitive value) are not join points, since Java requires their references to be inlined.]

Handler execution

When an exception handler executes. Handler execution join points are considered to have one argument, the exception being handled. No value is returned from a field set join point, so its return type is considered to be void.

Advice execution

When the body of code for a piece of advice executes.

B.3 Pointcuts

A pointcut is a program element that picks out join points and exposes data from the execution context of those join points. Pointcuts are used primarily by advice. They can be composed with boolean operators to build up other pointcuts. The primitive pointcuts and combinators provided by the language are:

`call (MethodPattern)`

Picks out each method call join point whose signature matches *MethodPattern*.

`execution (MethodPattern)`

Picks out each method execution join point whose signature matches

MethodPattern.

`get (FieldPattern)`

Picks out each field reference join point whose signature matches *FieldPattern*.

[Note that references to constant fields (static final fields bound to a constant string object or primitive value) are not join points, since Java requires them to be inlined.]

`set (FieldPattern)`

Picks out each field set join point whose signature matches *FieldPattern*. [Note that the initializations of constant fields (static final fields where the initializer is a constant string object or primitive value) are not join points, since Java requires their references to be inlined.]

`call (ConstructorPattern)`

Picks out each constructor call join point whose signature matches
ConstructorPattern.

execution(ConstructorPattern)
Picks out each constructor execution join point whose signature matches
ConstructorPattern.

initialization(ConstructorPattern)
Picks out each object initialization join point whose signature matches
ConstructorPattern.

preinitialization(ConstructorPattern)
Picks out each object pre-initialization join point whose signature matches
ConstructorPattern.

staticinitialization(TypePattern)
Picks out each static initializer execution join point whose signature matches
TypePattern.

handler(TypePattern)
Picks out each exception handler join point whose signature matches
TypePattern.

adviceexecution()
Picks out all advice execution join points.

within(TypePattern)
Picks out each join point where the executing code is defined in a type matched
by *TypePattern*.

withincode(MethodPattern)
Picks out each join point where the executing code is defined in a method whose
signature matches *MethodPattern*.

withincode(ConstructorPattern)
Picks out each join point where the executing code is defined in a constructor
whose signature matches *ConstructorPattern*.

cflow(Pointcut)
Picks out each join point in the control flow of any join point *P* picked out by
Pointcut, including *P* itself.

cflowbelow(Pointcut)
Picks out each join point in the control flow of any join point *P* picked out by
Pointcut, but not *P* itself.

this(Type or Id)
Picks out each join point where the currently executing object (the object bound
to *this*) is an instance of *Type*, or of the type of *Id* (which must be bound in the
enclosing advice or pointcut definition). Will not match any join points from
static contexts.

target(Type or Id)
Picks out each join point where the target object (the object on which a call or
field operation is applied to) is an instance of *Type*, or of the type of *Id* (which
must be bound in the enclosing advice or pointcut definition). Will not match any
calls, gets, or sets of static members.

args(Type or Id, ...)
Picks out each join point where the arguments are instances of a type of the
appropriate type pattern or identifier.

PointcutId(TypePattern or Id, ...)

Picks out each join point that is picked out by the user-defined pointcut designator named by *PointcutId*.

if (BooleanExpression)
Picks out each join point where the boolean expression evaluates to `true`. The boolean expression used can only access static members, variables exposed by the enclosing pointcut or advice, and `thisJoinPoint` forms. In particular, it cannot call non-static methods on the aspect.

! Pointcut
Picks out each join point that is not picked out by *Pointcut*.

Pointcut0 && Pointcut1
Picks out each join points that is picked out by both *Pointcut0* and *Pointcut1*.

Pointcut0 || Pointcut1
Picks out each join point that is picked out by either pointcuts. *Pointcut0* or *Pointcut1*.

(Pointcut)
Picks out each join points picked out by *Pointcut*.

B.3.1 Pointcut definition

Pointcuts are defined and named by the programmer with the `pointcut` declaration.

```
pointcut publicIntCall(int i):
    call(public * *(int)) && args(i);
```

A named pointcut may be defined in either a class or aspect, and is treated as a member of the class or aspect where it is found. As a member, it may have an access modifier such as `public` or `private`.

```
class C {
    pointcut publicCall(int i):
        call(public * *(int)) && args(i);
}

class D {
    pointcut myPublicCall(int i):
        C.publicCall(i) && within(SomeType);
}
```

Pointcuts that are not final may be declared abstract, and defined without a body. Abstract pointcuts may only be declared within abstract aspects.

```
abstract aspect A {
    abstract pointcut publicCall(int i);
}
```

In such a case, an extending aspect may override the abstract pointcut.

```
aspect B extends A {
    pointcut publicCall(int i): call(public Foo.m(int)) && args(i);
}
```

For completeness, a pointcut with a declaration may be declared `final`.

Though named pointcut declarations appear somewhat like method declarations, and can be overridden in subaspects, they cannot be overloaded. It is an error for two pointcuts to be named with the same name in the same class or aspect declaration.

The scope of a named pointcut is the enclosing class declaration. This is different than the scope of other members; the scope of other members is the enclosing class *body*. This means that the following code is legal:

```
aspect B percfow(publicCall()) {
    pointcut publicCall(): call(public Foo.m(int));
}
```

B.3.2 Context exposure

Pointcuts have an interface; they expose some parts of the execution context of the join points they pick out. For example, the `PublicIntCall` above exposes the first argument from the receptions of all public unary integer methods. This context is exposed by providing typed formal parameters to named pointcuts and advice, like the formal parameters of a Java method. These formal parameters are bound by name matching.

On the right-hand side of advice or pointcut declarations, in certain pointcut designators, a Java identifier is allowed in place of a type or collection of types. The pointcut designators that allow this are `this`, `target`, and `args`. In all such cases, using an identifier rather than a type does two things. First, it selects join points as based on the type of the formal parameter. So the pointcut

```
pointcut intArg(int i): args(i);
```

picks out join points where an `int` is being passed as an argument. Second, though, it makes the value of that argument available to the enclosing advice or pointcut.

Values can be exposed from named pointcuts as well, so

```
pointcut publicCall(int x): call(public *.*(int)) && intArg(x);
pointcut intArg(int i): args(i);
```

is a legal way to pick out all calls to public methods accepting an `int` argument, and exposing that argument.

There is one special case for this kind of exposure. Exposing an argument of type `Object` will also match primitive typed arguments, and expose a "boxed" version of the primitive. So,

```
pointcut publicCall(): call(public *.*(..)) && args(Object);
```

will pick out all unary methods that take, as their only argument, subtypes of `Object` (i.e., not primitive types like `int`), but

```
pointcut publicCall(Object o): call(public *.*(..)) && args(o);
```

will pick out all unary methods that take any argument. And if the argument was an `int`, then the value passed to advice will be of type `java.lang.Integer`.

B.3.3 Primitive pointcuts

B.3.3.1 Method-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture method call and execution join points.

- `call(MethodPattern)`
- `execution(MethodPattern)`

B.3.3.2 Field-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture field reference and set join points:

- `get(FieldPattern)`
- `set(FieldPattern)`

All set join points are treated as having one argument, the value the field is being set to, so at a set join point, that value can be accessed with an `args` pointcut. So an aspect guarding an integer variable `x` declared in type `T` might be written as

```
aspect GuardedX {
    static final int MAX_CHANGE = 100;
    before(int newval): set(int T.x) && args(newval) {
        if (Math.abs(newval - T.x) > MAX_CHANGE)
            throw new RuntimeException();
    }
}
```

B.3.3.3 Object creation-related pointcuts

AspectJ provides primitive pointcut designators designed to capture the initializer execution join points of objects.

- `call(ConstructorPattern)`
- `execution(ConstructorPattern)`
- `initialization(ConstructorPattern)`
- `preinitialization(ConstructorPattern)`

B.3.3.4 Class initialization-related pointcuts

AspectJ provides one primitive pointcut designator to pick out static initializer execution join points.

- `staticinitialization(TypePattern)`

B.3.3.5 Exception handler execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of exception handlers:

- `handler(TypePattern)`

All handler join points are treated as having one argument, the value of the exception being handled. That value can be accessed with an `args` pointcut. So an aspect used to put `FooException` objects into some normal form before they are handled could be written as

```
aspect NormalizeFooException {
    before(FooException e): handler(FooException) && args(e) {
        e.normalize();
    }
}
```

B.3.3.6 Advice execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of advice

- `adviceexecution()`

This can be used, for example, to filter out any join point in the control flow of advice from a particular aspect.

```
aspect TraceStuff {
    pointcut myAdvice(): adviceexecution() && within(TraceStuff);

    before(): call(* *(..)) && !cflow(myAdvice) {
        // do something
    }
}
```

B.3.3.7 State-based pointcuts

Many concerns cut across the dynamic times when an object of a particular type is executing, being operated on, or being passed around. AspectJ provides primitive pointcuts that capture join points at these times. These pointcuts use the dynamic types of their objects to pick out join points. They may also be used to expose the objects used for discrimination.

- `this(Type or Id)`
- `target(Type or Id)`

The `this` pointcut picks out each join point where the currently executing object (the object bound to `this`) is an instance of a particular type. The `target` pointcut picks out each join point where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type. Note that `target` should be understood to be the object the current join point is transferring control to. This means that the target object is the same as the current object at a method execution join point, for example, but may be different at a method call join point.

- `args(Type or Id or "..", ...)`

The `args` pointcut picks out each join point where the arguments are instances of some types. Each element in the comma-separated list is one of four things. If it is a type name, then the argument in that position must be an instance of that type. If it is an identifier, then that identifier must be bound in the enclosing advice or pointcut declaration, and so the argument in that position must be an instance of the type of the identifier (or of any type if the identifier is typed to `Object`). If it is the `"*"` wildcard, then any argument will match, and if it is the special wildcard `".."`, then any number of arguments will match, just like in signature patterns. So the pointcut

```
args(int, .., String)
```

will pick out all join points where the first argument is an `int` and the last is a `String`.

B.3.3.8 Control flow-based pointcuts

Some concerns cut across the control flow of the program. The `cflow` and `cflowbelow` primitive pointcut designators capture join points based on control flow.

- `cflow(Pointcut)`
- `cflowbelow(Pointcut)`

The `cflow` pointcut picks out all join points that occur between entry and exit of each join point *P* picked out by *Pointcut*, including *P* itself. Hence, it picks out the join points *in* the control flow of the join points picked out by *Pointcut*.

The `cflowbelow` pointcut picks out all join points that occur between entry and exit of each join point *P* picked out by *Pointcut*, but not including *P* itself. Hence, it picks out the join points *below* the control flow of the join points picked out by *Pointcut*.

B.3.3.9 Program text-based pointcuts

While many concerns cut across the runtime structure of the program, some must deal with the lexical structure. AspectJ allows aspects to pick out join points based on where their associated code is defined.

- `within(TypePattern)`
- `withincode(MethodPattern)`
- `withincode(ConstructorPattern)`

The `within` pointcut picks out each join point where the code executing is defined in the declaration of one of the types in *TypePattern*. This includes the class initialization, object initialization, and method and constructor execution join points for the type, as well as any join points associated with the statements and expressions of the type. It also includes any join points that are associated with code in a type's nested types, and that type's default constructor, if there is one.

The `withincode` pointcuts picks out each join point where the code executing is defined in the declaration of a particular method or constructor. This includes the method or constructor execution join point as well as any join points associated with the statements

and expressions of the method or constructor. It also includes any join points that are associated with code in a method or constructor's local or anonymous types.

B.3.3.10 Expression-based pointcuts

- `if(BooleanExpression)`

The `if` pointcut picks out join points based on a dynamic property. It's syntax takes an expression, which must evaluate to a boolean true or false. Within this expression, the `thisJoinPoint` object is available. So one (extremely inefficient) way of picking out all call join points would be to use the pointcut

```
if(thisJoinPoint.getKind().equals("call"))
```

B.3.4 Signatures

One very important property of a join point is its signature, which is used by many of AspectJ's pointcut designators to select particular join points.

B.3.4.1 Methods

Join points associated with methods typically have method signatures, consisting of a method name, parameter types, return type, the types of the declared (checked) exceptions, and some type that the method could be called on (below called the "qualifying type").

At a method call join point, the signature is a method signature whose qualifying type is the static type used to *access* the method. This means that the signature for the join point created from the call `((Integer)i).toString()` is different than that for the call `((Object)i).toString()`, even if `i` is the same variable.

At a method execution join point, the signature is a method signature whose qualifying type is the declaring type of the method.

B.3.4.2 Fields

Join points associated with fields typically have field signatures, consisting of a field name and a field type. A field reference join point has such a signature, and no parameters. A field set join point has such a signature, but has a single parameter whose type is the same as the field type.

B.3.4.3 Constructors

Join points associated with constructors typically have constructor signatures, consisting of a parameter types, the types of the declared (checked) exceptions, and the declaring type.

At a constructor call join point, the signature is the constructor signature of the called constructor. At a constructor execution join point, the signature is the constructor signature of the currently executing constructor.

At object initialization and pre-initialization join points, the signature is the constructor signature for the constructor that started this initialization: the first constructor entered during this type's initialization of this object.

B.3.4.4 Others

At a handler execution join point, the signature is composed of the exception type that the handler handles.

At an advice execution join point, the signature is composed of the aspect type, the parameter types of the advice, the return type (void for all but around advice) and the types of the declared (checked) exceptions.

B.3.5 Matching

The `withincode`, `call`, `execution`, `get`, and `set` primitive pointcut designators all use signature patterns to determine the join points they describe. A signature pattern is an abstract description of one or more join-point signatures. Signature patterns are intended to match very closely the same kind of things one would write when defining individual methods and constructors.

Method definitions in Java include method names, method parameters, return types, modifiers like `static` or `private`, and throws clauses, while constructor definitions omit the return type and replace the method name with the class name. The start of a particular method definition, in class `Test`, for example, might be

```
class C {  
    public final void foo() throws ArrayOutOfBoundsException { ... }  
}
```

In AspectJ, method signature patterns have all these, but most elements can be replaced by wildcards. So

```
call(public final void C.foo() throws ArrayOutOfBoundsException)
```

picks out call join points to that method, and the pointcut

```
call(public final void *.*() throws ArrayOutOfBoundsException)
```

picks out all call join points to methods, regardless of their name or which class they are defined on, so long as they take no arguments, return no value, are both `public` and `final`, and are declared to throw `ArrayOutOfBoundsException` exceptions.

The defining type name, if not present, defaults to `*`, so another way of writing that pointcut would be

```
call(public final void *() throws ArrayOutOfBoundsException)
```

Formal parameter lists can use the wildcard `..` to indicate zero or more arguments, so

```
execution(void m(..))
```

picks out execution join points for void methods named `m`, of any number of arguments, while

```
execution(void m(.., int))
```

picks out execution join points for void methods named `m` whose last parameter is of type `int`.

The modifiers also form part of the signature pattern. If an AspectJ signature pattern should match methods without a particular modifier, such as all non-public methods, the appropriate modifier should be negated with the `!` operator. So,

```
withincode(!public void foo())
```

picks out all join points associated with code in null non-public void methods named `foo`, while

```
withincode(void foo())
```

picks out all join points associated with code in null void methods named `foo`, regardless of access modifier.

Method names may contain the `*` wildcard, indicating any number of characters in the method name. So

```
call(int *())
```

picks out all call join points to `int` methods regardless of name, but

```
call(int get*())
```

picks out all call join points to `int` methods where the method name starts with the characters "get".

AspectJ uses the `new` keyword for constructor signature patterns rather than using a particular class name. So the execution join points of private null constructor of a class `C` defined to throw an `ArithmeticException` can be picked out with

```
execution(private C.new() throws ArithmeticException)
```

B.3.5.1 Matching based on the throws clause

Type patterns may be used to pick out methods and constructors based on their throws clauses. This allows the following two kinds of extremely wildcarded pointcuts:

```
pointcut throwsMathlike():
    // each call to a method with a throws clause containing at least
    // one exception exception with "Math" in its name.
    call(* *(..) throws *.*Math*);
```

```
pointcut doesNotThrowMathlike():
    // each call to a method with a throws clause containing no
    // exceptions with "Math" in its name.
    call(* *(..) throws !*.*Math*);
```

A *ThrowsClausePattern* is a comma-separated list of *ThrowsClausePatternItems*, where

ThrowsClausePatternItem:

[!] *TypeNamePattern*

A *ThrowsClausePattern* matches the throws clause of any code member signature. To match, each *ThrowsClausePatternItem* must match the throws clause of the member in question. If any item doesn't match, then the whole pattern doesn't match.

If a *ThrowsClausePatternItem* begins with "!", then it matches a particular throws clause if and only if *none* of the types named in the throws clause is matched by the *TypeNamePattern*.

If a *ThrowsClausePatternItem* does not begin with "!", then it matches a throws clause if and only if *any* of the types named in the throws clause is matched by the *TypeNamePattern*.

The rule for "!" matching has one potentially surprising property, in that these two pointcuts

- `call(* *(..) throws !IOException)`
- `call(* *(..) throws (!IOException))`

will match differently on calls to

```
void m() throws RuntimeException, IOException {}
```

[1] will NOT match the method `m()`, because method `m`'s throws clause declares that it throws `IOException`. [2] WILL match the method `m()`, because method `m`'s throws clause declares the it throws some exception which does not match `IOException`, i.e. `RuntimeException`.

B.3.6 Type patterns

Type patterns are a way to pick out collections of types and use them in places where you would otherwise use only one type. The rules for using type patterns are simple.

B.3.6.1 Type name patterns

First, all type names are also type patterns. So `Object`, `java.util.HashMap`, `Map.Entry`, `int` are all type patterns.

There is a special type name, `*`, which is also a type pattern. `*` picks out all types, including primitive types. So

```
call(void foo(*))
```

picks out all call join points to void methods named `foo`, taking one argument of any type.

Type names that contain the two wildcards `"*"` and `". ."` are also type patterns. The `*` wildcard matches zero or more characters characters except for `"."`, so it can be used when types have a certain naming convention. So

```
handler(java.util.*Map)
```

picks out the types `java.util.Map` and `java.util.HashMap`, among others, and

```
handler(java.util.*)
```

picks out all types that start with `"java.util."` and don't have any more `"."`s, that is, the types in the `java.util` package, but not inner types (such as `java.util.Map.Entry`).

The `".."` wildcard matches any sequence of characters that start and end with a `"."`, so it can be used to pick out all types in any subpackage, or all inner types. So

```
within(com.xerox..*)
```

picks out all join points where the code is in any type definition of a type whose name begins with `"com.xerox."`.

B.3.6.2 Subtype patterns

It is possible to pick out all subtypes of a type (or a collection of types) with the `"+"` wildcard. The `"+"` wildcard follows immediately a type name pattern. So, while

```
call(Foo.new())
```

picks out all constructor call join points where an instance of exactly type `Foo` is constructed,

```
call(Foo+.new())
```

picks out all constructor call join points where an instance of any subtype of `Foo` (including `Foo` itself) is constructed, and the unlikely

```
call(*Handler+.new())
```

picks out all constructor call join points where an instance of any subtype of any type whose name ends in `"Handler"` is constructed.

B.3.6.3 Array type patterns

A type name pattern or subtype pattern can be followed by one or more sets of square brackets to make array type patterns. So `Object[]` is an array type pattern, and so is `com.xerox..*[][]`, and so is `Object+[]`.

B.3.6.4 Type patterns

Type patterns are built up out of type name patterns, subtype patterns, and array type patterns, and constructed with boolean operators `&&`, `||`, and `!`. So

```
staticinitialization(Foo || Bar)
```

picks out the static initializer execution join points of either `Foo` or `Bar`, and

```
call((Foo+ && ! Foo).new(..))
```

picks out the constructor call join points when a subtype of `Foo`, but not `Foo` itself, is constructed.

B.4 Advice

Each piece of advice is of the form

```
[ strictfp ] AdviceSpec [ throws TypeList ] : Pointcut {
  Body }
```

where *AdviceSpec* is one of

- `before(Formals)`
- `after(Formals) returning [(Formal)]`
- `after(Formals) throwing [(Formal)]`
- `after(Formals)`
- `Type around(Formals)`

Advice defines crosscutting behavior. It is defined in terms of pointcuts. The code of a piece of advice runs at every join point picked out by its pointcut. Exactly how the code runs depends on the kind of advice.

AspectJ supports three kinds of advice. The kind of advice determines how it interacts with the join points it is defined over. Thus AspectJ divides advice into that which runs before its join points, that which runs after its join points, and that which runs in place of (or "around") its join points.

While before advice is relatively unproblematic, there can be three interpretations of after advice: After the execution of a join point completes normally, after it throws an exception, or after it does either one. AspectJ allows after advice for any of these situations.

```
aspect A {
  pointcut publicCall(): call(public Object *(..));
  after() returning (Object o): publicCall() {
    System.out.println("Returned normally with " + o);
  }
  after() throwing (Exception e): publicCall() {
    System.out.println("Threw an exception: " + e);
  }
  after(): publicCall(){
    System.out.println("Returned or threw an Exception");
  }
}
```

After returning advice may not care about its returned object, in which case it may be written

```
after() returning: call(public Object *(..)) {
  System.out.println("Returned normally");
}
```

It is an error to try to put after returning advice on a join point that does not return the correct type. For example,

```
after() returning (byte b): call(int String.length()) {
  // this is an error
}
```

is not allowed. But if no return value is exposed, or the exposed return value is typed to `Object`, then it may be applied to any join point. If the exposed value is typed to `Object`,

then the actual return value is converted to an object type for the body of the advice: `int` values are represented as `java.lang.Integer` objects, etc, and no value (from `void` methods, for example) is represented as `null`.

Around advice runs in place of the join point it operates over, rather than before or after it. Because around is allowed to return a value, it must be declared with a return type, like a method. A piece of around advice may be declared `void`, in which case it is not allowed to return a value, and instead whatever value the join point returned will be returned by the around advice (unless the around advice throws an exception of its own).

Thus, a simple use of around advice is to make a particular method constant:

```
aspect A {
    int around(): call(int C.foo()) {
        return 3;
    }
}
```

Within the body of around advice, though, the computation of the original join point can be executed with the special syntax

```
    proceed( ... )
```

The `proceed` form takes as arguments the context exposed by the around's pointcut, and returns whatever the around is declared to return. So the following around advice will double the second argument to `foo` whenever it is called, and then halve its result:

```
aspect A {
    int around(int i): call(int C.foo(Object, int)) && args(i) {
        int newi = proceed(i*2)
        return newi/2;
    }
}
```

If the return value of around advice is typed to `Object`, then the result of `proceed` is converted to an object representation, even if it is originally a primitive value. And when the advice returns an `Object` value, that value is converted back to whatever representation it was originally. So another way to write the doubling and halving advice is:

```
aspect A {
    Object around(int i): call(int C.foo(Object, int)) && args(i) {
        Integer newi = (Integer) proceed(i*2)
        return new Integer(newi.intValue() / 2);
    }
}
```

In all kinds of advice, the parameters of the advice behave exactly like method parameters. In particular, assigning to any parameter affects only the value of the parameter, not the value that it came from. This means that

```
aspect A {
    after() returning (int i): call(int C.foo()) {
        i = i * 2;
    }
}
```



```
    }
}
```

will *not* double the returned value of the advice. Rather, it will double the local parameter. Changing the values of parameters or return values of join points can be done by using around advice.

B.4.1 Advice modifiers

The `strictfp` modifier is the only modifier allowed on advice, and it has the effect of making all floating-point expressions within the advice be FP-strict.

B.4.2 Advice and checked exceptions

An advice declaration must include a `throws` clause listing the checked exceptions the body may throw. This list of checked exceptions must be compatible with each target join point of the advice, or an error is signalled by the compiler.

For example, in the following declarations:

```
import java.io.FileNotFoundException;

class C {
    int i;

    int getI() { return i; }
}

aspect A {
    before(): get(int C.i) {
        throw new FileNotFoundException();
    }
    before() throws FileNotFoundException: get(int C.i) {
        throw new FileNotFoundException();
    }
}
```

both pieces of advice are illegal. The first because the body throws an undeclared checked exception, and the second because field `get` join points cannot throw `FileNotFoundException`s.

The exceptions that each kind of join point in `AspectJ` may throw are:

method call and execution

the checked exceptions declared by the target method's `throws` clause.

constructor call and execution

the checked exceptions declared by the target constructor's `throws` clause.

field `get` and `set`

no checked exceptions can be thrown from these join points.

exception handler execution

the exceptions that can be thrown by the target exception handler.

static initializer execution

no checked exceptions can be thrown from these join points.

pre-initialization and initialization

any exception that is in the throws clause of *all* constructors of the initialized class.

advice execution

any exception that is in the throws clause of the advice.

B.4.3 Advice precedence

Multiple pieces of advice may apply to the same join point. In such cases, the resolution order of the advice is based on advice precedence.

B.4.3.1 Determining precedence

There are a number of rules that determine whether a particular piece of advice has precedence over another when they advise the same join point.

If the two pieces of advice are defined in different aspects, then there are three cases:

- If aspect A is matched earlier than aspect B in some `declare precedence` form, then all advice in concrete aspect A has precedence over all advice in concrete aspect B when they are on the same join point.
- Otherwise, if aspect A is a subaspect of aspect B, then all advice defined in A has precedence over all advice defined in B. So, unless otherwise specified with `declare precedence`, advice in a subaspect has precedence over advice in a superaspect.
- Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.

If the two pieces of advice are defined in the same aspect, then there are two cases:

- If either are `after` advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.

These rules can lead to circularity, such as

```
aspect A {  
    before(): execution(void main(String[] args)) {}  
    after():  execution(void main(String[] args)) {}  
    before(): execution(void main(String[] args)) {}  
}
```

such circularities will result in errors signalled by the compiler.

B.4.3.2 Effects of precedence

At a particular join point, advice is ordered by precedence.

A piece of `around` advice controls whether advice of lower precedence will run by calling `proceed`. The call to `proceed` will run the advice with next precedence, or the computation under the join point if there is no further advice.

A piece of `before` advice can prevent advice of lower precedence from running by throwing an exception. If it returns normally, however, then the advice of the next precedence, or the computation under the join point if there is no further advice, will run.

Running `after returning` advice will run the advice of next precedence, or the computation under the join point if there is no further advice. Then, if that computation returned normally, the body of the advice will run.

Running `after throwing` advice will run the advice of next precedence, or the computation under the join point if there is no further advice. Then, if that computation threw an exception of an appropriate type, the body of the advice will run.

Running `after` advice will run the advice of next precedence, or the computation under the join point if there is no further advice. Then the body of the advice will run.

B.4.4 Reflective access to the join point

Three special variables are visible within bodies of advice: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`. Each is bound to an object that encapsulates some of the context of the advice's current or enclosing join point. These variables exist because some pointcuts may pick out very large collections of join points. For example, the pointcut

```
pointcut publicCall(): call(public * *(..));
```

picks out calls to many methods. Yet the body of advice over this pointcut may wish to have access to the method name or parameters of a particular join point.

`thisJoinPoint` is bound to a complete join point object.

`thisJoinPointStaticPart` is bound to a part of the join point object that includes less information, but for which no memory allocation is required on each execution of the advice. It is equivalent to `thisJoinPoint.getStaticPart()`.

`thisEnclosingJoinPointStaticPart` is bound to the static part of the join point enclosing the current join point. Only the static part of this enclosing join point is available through this mechanism.

Standard Java reflection uses objects from the `java.lang.reflect` hierarchy to build up its reflective objects. Similarly, AspectJ join point objects have types in a type hierarchy. The type of objects bound to `thisJoinPoint` is `org.aspectj.lang.JoinPoint`, while `thisStaticJoinPoint` is bound to objects of interface type `org.aspectj.lang.JoinPoint.StaticPart`.

B.5 Static crosscutting

Advice declarations change the behavior of classes they crosscut, but do not change their static type structure. For crosscutting concerns that do operate over the static structure of type hierarchies, AspectJ provides inter-type member declarations and other `declare` forms.

B.5.1 Inter-type member declarations

AspectJ allows the declaration of members by aspects that are associated with other types.

An inter-type method declaration looks like

- `[Modifiers] Type OnType . Id(Formals) [ThrowsClause] { Body }`
- `abstract [Modifiers] Type OnType . Id(Formals) [ThrowsClause] ;`

The effect of such a declaration is to make *OnType* support the new method. Even if *OnType* is an interface. Even if the method is neither public nor abstract. So the following is legal AspectJ code:

```
interface Iface {}

aspect A {
    private void Iface.m() {
        System.err.println("I'm a private method on an interface");
    }
    void worksOnI(Iface iface) {
        // calling a private method on an interface
        iface.m();
    }
}
```

An inter-type constructor declaration looks like

- `[Modifiers] OnType . new (Formals) [ThrowsClause] { Body }`

The effect of such a declaration is to make *OnType* support the new constructor. It is an error for *OnType* to be an interface.

Note that in the Java language, classes that define no constructors have an implicit no-argument constructor that just calls `super()`. This means that attempting to declare a no-argument inter-type constructor on such a class may result in a conflict, even though it *looks* like no constructor is defined.

An inter-type field declaration looks like one of

- `[Modifiers] Type OnType . Id = Expression;`
- `[Modifiers] Type OnType . Id;`

The effect of such a declaration is to make *OnType* support the new field. Even if *OnType* is an interface. Even if the field is neither public, nor static, nor final.

The initializer, if any, of an inter-type field definition runs before the class-local initializers defined in its target class.

Any occurrence of the identifier `this` in the body of an inter-type constructor or method declaration, or in the initializer of an inter-type field declaration, refers to the *OnType* object rather than to the aspect type; it is an error to access `this` in such a position from a `static` inter-type member declaration.

B.5.2 Access modifiers

Inter-type member declarations may be public or private, or have default (package-protected) visibility. AspectJ does not provide protected inter-type members.

The access modifier applies in relation to the aspect, not in relation to the target type. So a private inter-type member is visible only from code that is defined within the declaring aspect. A default-visibility inter-type member is visible only from code that is defined within the declaring aspect's package.

Note that a declaring a private inter-type method (which AspectJ supports) is very different from inserting a private method declaration into another class. The former allows access only from the declaring aspect, while the latter would allow access only from the target type. Java serialization, for example, uses the presense of a private method `void writeObject(ObjectOutputStream)` for the implementation of `java.io.Serializable`. A private inter-type declaration of that method would not fulfill this requirement, since it would be private to the aspect, not private to the target type.

B.5.3 Conflicts

Inter-type declarations raise the possibility of conflicts among locally declared members and inter-type members. For example, assuming `otherPackage` is not the package containing the aspect `A`, the code

```
aspect A {
    private Registry otherPackage.*.r;
    public void otherPackage.*.register(Registry r) {
        r.register(this);
        this.r = r;
    }
}
```

declares that every type in `otherPackage` has a field `r`. This field, however, is only accessible from the code inside of aspect `A`. The aspect also declares that every type in `otherPackage` has a method `"register"`, but makes this method accessible from everywhere.

If any type in `otherPackage` already defines a private or package-protected field `"r"`, there is no conflict: The aspect cannot see such a field, and no code in `otherPackage` can see the inter-type `"r"`.

If any type in `otherPackage` defines a public field `"r"`, there is a conflict: The expression

```
this.r = r
```

is an error, since it is ambiguous whether the private inter-type "r" or the public locally-defined "r" should be used.

If any type in `otherPackage` defines any method "register(Registry)" there is a conflict, since it would be ambiguous to any code that could see such a defined method which "register(Registry)" method was applicable.

Conflicts are resolved as much as possible as per Java's conflict resolution rules:

- A subclass can inherit multiple *fields* from its superclasses, all with the same name and type. However, it is an error to have an ambiguous *reference* to a field.
- A subclass can only inherit multiple *methods* with the same name and argument types from its superclasses if only zero or one of them is concrete (i.e., all but one is abstract, or all are abstract).

Given a potential conflict between inter-type member declarations in different aspects, if one aspect has precedence over the other its declaration will take effect without any conflict notice from compiler. This is true both when the precedence is declared explicitly with `declare precedence` as well as when when sub-aspects implicitly have precedence over their super-aspect.

B.5.4 Extension and Implementation

An aspect may change the inheritance hierarchy of a system by changing the superclass of a type or adding a superinterface onto a type, with the `declare parents` form.

- `declare parents: TypePattern extends Type;`
- `declare parents: TypePattern implements TypeList;`

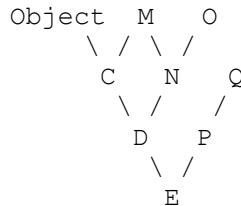
For example, if an aspect wished to make a particular class runnable, it might define appropriate inter-type `void run()` method, but it should also declare that the class fulfills the `Runnable` interface. In order to implement the methods in the `Runnable` interface, the inter-type `run()` method must be public:

```
aspect A {  
    declare parents: SomeClass implements Runnable;  
    public void SomeClass.run() { ... }  
}
```

B.5.5 Interfaces with members

Through the use of inter-type members, interfaces may now carry (non-public-static-final) fields and (non-public-abstract) methods that classes can inherit. Conflicts may occur from ambiguously inheriting members from a superclass and multiple superinterfaces.

Because interfaces may carry non-static initializers, each interface behaves as if it has a zero-argument constructor containing its initializers. The order of super-interface instantiation is observable. We fix this order with the following properties: A supertype is initialized before a subtype, initialized code runs only once, and the initializers for a type's superclass are run before the initializers for its superinterfaces. Consider the following hierarchy where {Object, C, D, E} are classes, {M, N, O, P, Q} are interfaces.



when a new E is instantiated, the initializers run in this order:

```
Object M C O N D Q P E
```

B.5.6 Warnings and Errors

An aspect may specify that a particular join point should never be reached.

- declare error: *Pointcut*: *String*;
- declare warning: *Pointcut*: *String*;

If the compiler determines that a join point in *Pointcut* could possibly be reached, then it will signal either an error or warning, as declared, using the *String* for its message.

B.5.7 Softened exceptions

An aspect may specify that a particular kind of exception, if thrown at a join point, should bypass Java's usual static exception checking system and instead be thrown as a `org.aspectj.lang.SoftException`, which is subtype of `RuntimeException` and thus does not need to be declared.

- declare soft: *Type*: *Pointcut*;

For example, the aspect

```
aspect A {
    declare soft: Exception: execution(void main(String[] args));
}
```

Would, at the execution join point, catch any `Exception` and rethrow a `org.aspectj.lang.SoftException` containing original exception.

This is similar to what the following advice would do

```
aspect A {
```

```

        void around() execution(void main(String[] args)) {
            try { proceed(); }
            catch (Exception e) {
                throw new org.aspectj.lang.SoftException(e);
            }
        }
    }
}

```

except, in addition to wrapping the exception, it also affects Java's static exception checking mechanism.

B.5.8 Advice Precedence

An aspect may declare a precedence relationship between concrete aspects with the `declare precedence` form:

- `declare precedence : TypePatternList ;`

This signifies that if any join point has advice from two concrete aspects matched by some pattern in *TypePatternList*, then the precedence of the advice will be the order of in the list.

In *TypePatternList*, the wildcard "*" can appear at most once, and it means "any type not matched by any other pattern in the list".

For example, the constraints that (1) aspects that have Security as part of their name should have precedence over all other aspects, and (2) the Logging aspect (and any aspect that extends it) should have precedence over all non-security aspects, can be expressed by:

```
declare precedence: *.*Security*, Logging+, *;
```

For another example, the CountEntry aspect might want to count the entry to methods in the current package accepting a Type object as its first argument. However, it should count all entries, even those that the aspect DisallowNulls causes to throw exceptions. This can be accomplished by stating that CountEntry has precedence over DisallowNulls. This declaration could be in either aspect, or in another, ordering aspect:

```

aspect Ordering {
    declare precedence: CountEntry, DisallowNulls;
}
aspect DisallowNulls {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj,
..);
    before(Type obj): allTypeMethods(obj) {
        if (obj == null) throw new RuntimeException();
    }
}
aspect CountEntry {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj,
..);
    static int count = 0;
    before(): allTypeMethods(Type) {

```



```

        count++;
    }
}

```

B.5.8.1 Various cycles

It is an error for any aspect to be matched by more than one `TypePattern` in a single declare precedence, so:

```
declare precedence: A, B, A ; // error

```

However, multiple declare precedence forms may legally have this kind of circularity. For example, each of these declare precedence is perfectly legal:

```
declare precedence: B, A;
declare precedence: A, B;

```

And a system in which both constraints are active may also be legal, so long as advice from A and B don't share a join point. So this is an idiom that can be used to enforce that A and B are strongly independent.

B.5.8.2 Applies to concrete aspects

Consider the following library aspects:

```

abstract aspect Logging {
    abstract pointcut logged();

    before(): logged() {
        System.err.println("thisJoinPoint: " + thisJoinPoint);
    }
}

aspect aspect MyProfiling {
    abstract pointcut profiled();

    Object around(): profiled() {
        long beforeTime = System.currentTimeMillis();
        try {
            return proceed();
        } finally {
            long afterTime = System.currentTimeMillis();
            addToProfile(thisJoinPointStaticPart,
                        afterTime - beforeTime);
        }
    }

    abstract void addToProfile(
        org.aspectj.JoinPoint.StaticPart jp,
        long elapsed);
}

```

In order to use either aspect, they must be extended with concrete aspects, say, `MyLogging` and `MyProfiling`. Because advice only applies from concrete aspects, the declare precedence form only matters when declaring precedence with concrete aspects. So

```
declare precedence: Logging, Profiling;
```

has no effect, but both

```
declare precedence: MyLogging, MyProfiling;
```

```
declare precedence: Logging+, Profiling+;
```

are meaningful.

B.5.9 Statically determinable pointcuts

Pointcuts that appear inside of `declare` forms have certain restrictions. Like other pointcuts, these pick out join points, but they do so in a way that is statically determinable.

Consequently, such pointcuts may not include, directly or indirectly (through user-defined pointcut declarations) pointcuts that discriminate based on dynamic (runtime) context. Therefore, such pointcuts may not be defined in terms of

- `cflow`
- `cflowbelow`
- `this`
- `target`
- `args`
- `if`

all of which can discriminate on runtime information.

B.6 Aspects

An aspect is a crosscutting type defined by the `aspect` declaration. The `aspect` declaration is similar to the `class` declaration in that it defines a type and an implementation for that type. It differs in that the type and implementation can cut across other types (including those defined by other aspect declarations), and that it may not be directly instantiated with a new expression, with cloning, or with serialization. Aspects may have one constructor definition, but if so it must be of a constructor taking no arguments and throwing no checked exceptions.

Aspects may be defined either at the package level, or as a static nested aspect -- that is, a static member of a class, interface, or aspect. If it is not at the package level, the aspect *must* be defined with the `static` keyword. Local and anonymous aspects are not allowed.

B.6.1 Aspect Extension

To support abstraction and composition of crosscutting concerns, aspects can be extended in much the same way that classes can. Aspect extension adds some new rules, though.

B.6.1.1 Aspects may extend classes and implement interfaces

An aspect, abstract or concrete, may extend a class and may implement a set of interfaces. Extending a class does not provide the ability to instantiate the aspect with a new expression: The aspect may still only define a null constructor.

B.6.1.2 Classes may not extend aspects

It is an error for a class to extend or implement an aspect.

B.6.1.3 Aspects extending aspects

Aspects may extend other aspects, in which case not only are fields and methods inherited but so are pointcuts. However, aspects may only extend abstract aspects. It is an error for a concrete aspect to extend another concrete aspect.

B.6.2 Aspect instantiation

Unlike class expressions, aspects are not instantiated with `new` expressions. Rather, aspect instances are automatically created to cut across programs.

Because advice only runs in the context of an aspect instance, aspect instantiation indirectly controls when advice runs.

The criteria used to determine how an aspect is instantiated is inherited from its parent aspect. If the aspect has no parent aspect, then by default the aspect is a singleton aspect.

B.6.2.1 Singleton Aspects

- `aspect Id { ... }`
- `aspect Id issingleton { ... }`

By default (or by using the modifier `issingleton`) an aspect has exactly one instance that cuts across the entire program. That instance is available at any time during program execution with the static method `aspectOf()` defined on the aspect -- so, in the above examples, `A.aspectOf()` will return A's instance. This aspect instance is created as the aspect's classfile is loaded.

Because the an instance of the aspect exists at all join points in the running of a program (once its class is loaded), its advice will have a chance to run at all such join points.

(In actuality, one instance of the aspect A is made for each version of the aspect A, so there will be one instantiation for each time A is loaded by a different classloader.)

B.6.2.2 Per-object aspects

- `aspect Id perthis(Pointcut) { ... }`
- `aspect Id pertarget(Pointcut) { ... }`

If an aspect A is defined `perthis(Pointcut)`, then one object of type A is created for every object that is the executing object (i.e., "this") at any of the join points picked out

by *Pointcut*. The advice defined in A may then run at any join point where the currently executing object has been associated with an instance of A.

Similarly, if an aspect A is defined `per target (Pointcut)`, then one object of type A is created for every object that is the target object of the join points picked out by *Pointcut*. The advice defined in A may then run at any join point where the target object has been associated with an instance of A.

In either case, the static method call `A.aspectOf(Object)` can be used to get the aspect instance (of type A) registered with the object. Each aspect instance is created as early as possible, but not before reaching a join point picked out by *Pointcut* where there is no associated aspect of type A.

Both `per this` and `per target` aspects may be affected by code the AspectJ compiler controls, as discussed in the [Implementation Limitations](#) appendix.

B.6.2.3 Per-control-flow aspects

- `aspect Id per cflow(Pointcut) { ... }`
- `aspect Id per cflowbelow(Pointcut) { ... }`

If an aspect A is defined `per cflow(Pointcut)` or `per cflowbelow(Pointcut)`, then one object of type A is created for each flow of control of the join points picked out by *Pointcut*, either as the flow of control is entered, or below the flow of control, respectively. The advice defined in A may run at any join point in or under that control flow. During each such flow of control, the static method `A.aspectOf()` will return an object of type A. An instance of the aspect is created upon entry into each such control flow.

B.6.2.4 Aspect instantiation and advice

All advice runs in the context of an aspect instance, but it is possible to write a piece of advice with a pointcut that picks out a join point that must occur before aspect instantiation. For example:

```
public class Client
{
    public static void main(String[] args) {
        Client c = new Client();
    }
}

aspect Watchcall {
    pointcut myConstructor(): execution(new(..));

    before(): myConstructor() {
        System.err.println("Entering Constructor");
    }
}
```

The `before` advice should run before the execution of all constructors in the system. It must run in the context of an instance of the `Watchcall` aspect. The only way to get such

an instance is to have Watchcall's default constructor execute. But before that executes, we need to run the before advice...

There is no general way to detect these kinds of circularities at compile time. If advice runs before its aspect is instantiated, AspectJ will throw a [org.aspectj.lang.NoAspectBoundException](http://www.eclipse.org/aspectj/doc/inter/ajrt/NoAspectBoundException.html).

B.6.3 Aspect privilege

- `privileged aspect Id { ... }`

Code written in aspects is subject to the same access control rules as Java code when referring to members of classes or aspects. So, for example, code written in an aspect may not refer to members with default (package-protected) visibility unless the aspect is defined in the same package.

While these restrictions are suitable for many aspects, there may be some aspects in which advice or inter-type members needs to access private or protected resources of other types. To allow this, aspects may be declared `privileged`. Code in privileged aspects has access to all members, even private ones.

```
class C {
    private int i = 0;
    void incI(int x) { i = i+x; }
}
privileged aspect A {
    static final int MAX = 1000;
    before(int x, C c): call(void C.incI(int)) && target(c) &&
args(x) {
        if (c.i+x > MAX) throw new RuntimeException();
    }
}
```

In this case, if A had not been declared privileged, the field reference `c.i` would have resulted in an error signaled by the compiler.

If a privileged aspect can access multiple versions of a particular member, then those that it could see if it were not privileged take precedence. For example, in the code

```
class C {
    private int i = 0;
    void foo() { }
}
privileged aspect A {
    private int C.i = 999;
    before(C c): call(void C.foo()) target(c) {
        System.out.println(c.i);
    }
}
```

A's private inter-type field `C.i`, initially bound to 999, will be referenced in the body of the advice in preference to C's privately declared field, since the A would have access to its own inter-type fields even if it were not privileged.

Note that a privileged aspect can access private inter-type declarations made by other aspects, since they are simply considered private members of that other aspect.

Appendix C Implementation Limitations

The initial implementations of AspectJ have all been compiler-based implementations. Certain elements of AspectJ's semantics are difficult to implement without making modifications to the virtual machine, which a compiler-based implementation cannot do. One way to deal with this problem would be to specify only the behavior that is easiest to implement. We have chosen a somewhat different approach, which is to specify an ideal language semantics, as well as a clearly defined way in which implementations are allowed to deviate from that semantics. This makes it possible to develop conforming AspectJ implementations today, while still making it clear what later, and presumably better, implementations should do tomorrow.

According to the AspectJ language semantics, the declaration

```
before(): get(int Point.x) { System.out.println("got x"); }
```

should advise all accesses of a field of type `int` and name `x` from instances of type (or subtype of) `Point`. It should do this regardless of whether all the source code performing the access was available at the time the aspect containing this advice was compiled, whether changes were made later, etc.

But AspectJ implementations are permitted to deviate from this in a well-defined way -- they are permitted to advise only accesses in *code the implementation controls*. Each implementation is free within certain bounds to provide its own definition of what it means to control code.

In the current AspectJ compiler, `ajc`, control of the code means having bytecode for any aspects and all the code they should affect available during the compile. This means that if some class `Client` contains code with the expression `new Point().x` (which results in a field `get join point` at runtime), the current AspectJ compiler will fail to advise that access unless `Client.java` or `Client.class` is compiled as well. It also means that join points associated with code in native methods (including their execution join points) cannot be advised.

Different join points have different requirements. Method and constructor call join points can be advised only if `ajc` controls the bytecode for the caller. Field reference or assignment join points can be advised only if `ajc` controls the bytecode for the "caller", the code actually making the reference or assignment. Initialization join points can be advised only if `ajc` controls the bytecode of the type being initialized, and execution join points can be advised only if `ajc` controls the bytecode for the method or constructor body in question.

Aspects that are defined `perthis` or `pertarget` also have restrictions based on control of the code. In particular, at a join point where the bytecode for the currently executing object is not available, an aspect defined `perthis` of that join point will not be associated. So aspects defined `perthis(Object)` will not create aspect instances for every object unless `Object` is part of the compile. Similar restrictions apply to `pertarget` aspects.

Inter-type declarations such as `declare parents` also have restrictions based on control of the code. If the bytecode for the target of an inter-type declaration is not available, then

the inter-type declaration is not made on that target. So, declare `parents : String implements MyInterface` will not work for `java.lang.String` unless `java.lang.String` is part of the compile.

Other AspectJ implementations, indeed, future versions of ajc, may define *code the implementation controls* more liberally or restrictively.

The important thing to remember is that core concepts of AspectJ, such as the join point, are unchanged, regardless of which implementation is used. During your development, you will have to be aware of the limitations of the ajc compiler you're using, but these limitations should not drive the design of your aspects.

Appendix D Talks Presented

D.1 Tutorials

AOSD 2003

18 March, 2003, Erik Hilsdale and Wes Isberg, advanced

AOSD 2003

17 March, 2003, Erik Hilsdale and Wes Isberg, intro

ACM Chapter Meeting Boston, MA

November 16, 2002, Gregor Kiczales and Ron Bodkin and Erik Hilsdale, full-day

OOPSLA Seattle, WA

November 6, 2002, Erik Hilsdale and Jim Hugunin, full-day

IPSJ Japan

August 27, 2002, Gregor Kiczales

AOSD Conference

April 23, 2002, Erik Hilsdale and Jim Hugunin, advanced

AOSD Conference

April 23, 2002, Erik Hilsdale and Jim Hugunin, intro

AspectJ Workshop

January 11, 2002, whole team

OOPSLA Tampa Bay, Florida

October 15, 2001, Erik Hilsdale and Mik Kersten, full-day

Reflection Kyoto, Japan

September 24, 2001, Gregor Kiczales, half-day

ESEC/FSE Vienna, Austria

September 11, 2001, Gregor Kiczales, full-day

TOOLS USA Santa Barbara, CA

August 1, 2001, Erik Hilsdale, half-day

ECOOP Budapest, Hungary

June 20-21, 2001, Gregor Kiczales and Erik Hilsdale, full-day

Boeing St. Louis, MO

April 2001, Gregor Kiczales

FSE San Diego, CA

November 7, 2000, Gregor Kiczales

OOPSLA Minneapolis, MN

October 27, 2000, Gregor Kiczales and Erik Hilsdale, full-day

Erfurt, Germany

September 10, 2000, Cristina Lopes, tutorial and workshop

TOOLS USA Santa Barbara, CA

July 30, 2000, Cristina Lopes, tutorial

D.2 Talks, Demos, and Keynotes

SD East Boston, MA

November 20, 2002, Erik Hilsdale and Ron Bodkin

SD East Boston, MA
 November 18, 2002, Gregor Kiczales, keynote.
 Eclipse BoF at OOPSLA Seattle, WA
 November 5, 2002, Mik Kersten, AJDT report
 OOPSLA Seattle, WA
 November 7, 2002, Mik Kersten and Erik Hilsdale, demo
 OOPSLA Seattle, WA
 November 8, 2002, Mik Kersten and Erik Hilsdale, demo
 Net Object Days Erfurt, Germany
 October 10, 2002, Gregor Kiczales, keynote
 Music City Java Conference Nashville, TN
 September 27, 2002, Mik Kersten, talk
 IPSJ Japan
 August 27, 2002, Gregor Kiczales, keynote
 SD Forum Java SIG Palo Alto, CA
 September 3, 2002, Mik Kersten, talk
 SD West, San Jose, CA
 April 26, 2002 Ron Bodkin: "Aspect-Oriented Software Development with AspectJ"
 SD West, San Jose, CA
 April 26, 2002 Ron Bodkin: "Better Javatm Development with AspectJ".
 JavaOne, San Francisco, CA
 March 25-29, 2002. Jim Hugunin and Mik Kersten, BOF
 OOPSLA Tampa Bay, Florida
 October 18, 2001, Erik Hilsdale and Mik Kersten, demo
 OOPSLA Tampa Bay, Florida
 October 17, 2001, Erik Hilsdale and Mik Kersten, demo
 Reflection Kyoto, Japan
 September 25, 2001, Gregor Kiczales, keynote
 JAOO Aarhus, Denmark
 September 10, 2001, Gregor Kiczales, talk
 BBN Lecture Series Cambridge, MA
 August 14, 2001, Gregor Kiczales, talk
 San Jose Java SIG San Jose, CA
 August 8, 2001, Jim Hugunin, talk
 ECOOP Budapest, Hungary
 June 21, 2001, Erik Hilsdale, demo
 ECOOP Budapest, Hungary
 June 20, 2001, Erik Hilsdale, demo
 Rockwell Martin
 June 2001, Gregor Kiczales, talk
 University College Dublin Dublin, Ireland
 April 6, 2001, Erik Hilsdale, talk
 O'Reilly Conference on Enterprise Java Santa Clara, CA
 March 29, 2001, Jim Hugunin, talk
 SIGS JavaPlus San Jose, CA
 October 29, 2000, Erik Hilsdale and Mik Kersten, talk

OOPSLA Minneapolis, MN

October 27, 2000, Jim Hugunin and Mik Kersten, demo

JAOO Aarhus, Denmark

September 25, 2000, Gregor Kiczales, talk